JPL Publication 88-32

Concurrent Image Processing Executive (CIPE)

Meemong Lee
Gregory T. Cooper
Steven L. Groom
Alan S. Mazer
Winifred I. Williams

November 1988

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL Publication 88-32

Concurrent Image Processing Executive (CIPE)

Meemong Lee
Gregory T. Cooper
Steven L. Groom
Alan S. Mazer
Winifred I. Williams

November 1988

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

ABSTRACT

This report describes the design and implementation of a Concurrent Image Processing Executive (CIPE), which is intended to become the support system software for a prototype high performance science analysis workstation. The target machine for this software is a JPL/Caltech Mark IIIfp Hypercube hosted by either a MASSCOMP 5600 or a Sun-3, Sun-4 workstation; however, the design will accommodate other concurrent machines of similar architecture, *i.e.*, local memory, multiple-instruction-multiple-data (MIMD) machines. The CIPE system provides both a multimode user interface and an applications programmer interface, and has been designed around four loosely coupled modules: (1) user interface, (2) host-resident executive, (3) hypercube-resident executive, and (4) application functions. The loose coupling between modules allows modification of a particular module without significantly affecting the other modules in the system. In order to enhance hypercube memory utilization and to allow expansion of image processing capabilities, a specialized program management method, incremental loading, was devised. To minimize data transfer between host and hypercube, a data management method which distributes, redistributes, and tracks data set information was implemented. The data management also allows data sharing among application programs. The CIPE software architecture provides a flexible environment for scientific analysis of complex remote sensing image data, such as imaging spectrometry, utilizing state-of-the-art concurrent computation capabilities.

ACKNOWLEDGMENTS

The authors would like to acknowledge Dr. Jerry Solomon for his support and direction of this work. The rest of the authors would also like to thank Winifred Williams for her selfless effort in editing this report for publication.

The research described in this paper was carried out by the Observational Systems division of the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the National Aeronautics and Space Administration (NASA). Special thanks are due to the office of Space Science and Applications (Code EC) of NASA for their encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

The current and projected growth in volume and complexity of the National Aeronautics and Space Administration (NASA) mission remote sensing image data is placing a considerable strain on the computational resources available to the science community for analyzing and interpreting this valuable source of information. By the end of this century, given the Earth Observing System (EOS) and other NASA space exploration missions, the wealth of data available to scientists in a variety of disciplines will be truly astounding. In order to address the problems associated with analyzing this data, a new generation of computational resources and tools must be developed and made available to the NASA science community. This task report describes work carried out during FY88 which addresses one aspect of this problem, *i.e.*, the development of user environments and tools for high-performance science analysis workstations. This work has been carried out in the context of implementing a Concurrent Image Processing and Analysis Testbed (CIPAT), which will form a prototype high-performance workstation. The utilization of emerging concurrent processing technology, in particular multiple-instruction-multiple-data (MIMD) architectures which provide large amounts of local memory, is a natural approach to handling compute-intensive problems with large data sets. This technology presents its own unique set of problems, however, particularly with respect to system software design and interactive user environments. The work described in this report provides one solution to these problems through the implementation of a general executive software system, the Concurrent Image Processing Executive (CIPE).

As the name implies, CIPE was designed for hardware systems which utilize concurrent computational resources for the processing of image and other multidimensional data; more specifically, the concurrent computational resource is assumed to be an attached processor which is connected to a *host* computer. In this context, the host machine provides the usual broad range of operating system services, file I/O, display device hardware, network connections, *etc.* Although the executive described in this report could be extended to cover commercial *multiprocessor* computers such as Alliant, Convex, Elxsi, Flex, *etc.*, that was not a design objective. The hardware environment within which CIPE was developed consists of a MASSCOMP 5600 dual-processor host machine, with approximately 800 MBytes of disk storage, an International Imaging Systems (I^2S) IVAS image display processor, and an 8-node JPL/Caltech Mark IIIfp hypercube. The host machine will be changed to a Sun-4/260 during the first quarter of FY89 in order to provide a more flexible environment and to accommodate much higher data transfer rates utilizing new concurrent I/O capabilities.

CIPE was designed around four major software modules: (1) user interface, (2) host system monitor, (3) hypercube monitor, and (4) application functions. A major design objective was to provide a powerful and flexible user interface for

efficient utilization of the system's considerable computational resources. System modularity provides relatively easy functional extensibility, both at the user level and the applications programming level. CIPE provides a number of utilities which greatly ease the applications programmer's difficulties in dealing with a new machine architecture, *i.e.*, the hypercube topology. We believe that the system described in this report provides a flexible framework within which to begin building a powerful user-oriented computing environment for high-performance science analysis workstations.

This report is intended to serve both as an overview of the CIPE design and implementation philosophy as well as provide the level of detail necessary for both users and applications programmers to become familiar with the CIPE environment. Section 1 is a general overview of CIPE as an executive, including a description of its design objectives and the general implementation approach. Section 2 describes the three user interface modalities of command line interpreter, menu mode, and window mode. Section 3 describes the host system monitor functions with respect to generic operating system functions and concurrent system interface functions. Section 4 describes the hypercube monitor functions, including the interface mechanism to the host system and CIPE management of data and application programs. Section 5 describes how to write an application program in the CIPE environment, including utilization of user interface modes, host system monitor functions, and hypercube monitor functions. A set of built-in functions for image processing operations is also examined. Finally, Section 6 provides a look at future directions in CIPE development with respect to a general scientific computing environment for high-performance workstations.

1. CONCURRENT IMAGE PROCESSING EXECUTIVE (CIPE) OVERVIEW

CIPE was designed and implemented by the Image Analysis Systems Group for the purpose of utilizing various concurrent architectures in a high rate data processing environment. Concurrent systems provide greatly enhanced computational power by integrating large numbers of processors into systems via various interconnection topologies. However, they do present significant programming difficulties due to their architectural complexities. In particular, utilization of such systems for interactive image processing requires a unique kind of software environment which shields users and programmers from architectural complexities while offering the computational advantages of concurrent systems.

1.1. WHAT IS AN EXECUTIVE?

Though there is no written definition for an executive, in general, an executive is software which resides on top of an operating system and provides common resources to users and programmers within a well-defined application area. The common goal of an interactive image processing environment in a concurrent system configuration is the utilization of a concurrent architecture for faster processing of image data, visual presentation, and real-time user interaction. The shared resources include user interfaces, data manipulation, display, file I/O, and interfaces to concurrent systems. Therefore, an executive approach was chosen to provide a software environment (user and programmer) for concurrent image processing.

CIPE has distinctive characteristics as a concurrent system executive, as a high rate data processing executive, and as an interactive image processing executive. As a concurrent system executive, it provides a programming environment which shields a programmer from architectural complexities while it utilizes the maximum potential of a concurrent system. As a high rate data processing executive, it provides a flexible file management scheme and efficient data manipulation and representation mechanisms. As an interactive image processing executive, it offers a friendly, flexible, and efficient user environment.

1.2. DESIGN OBJECTIVES

As CIPE is an executive serving three distinctive purposes, utilization of a concurrent system, interactive image processing, and high rate data processing, it has three corresponding design objectives and one overall objective.

First, the design objective as a concurrent system executive was to utilize a concurrent system to its full potential. The maximum concurrency of the system for data processing was pursued via devising appropriate data and program management schemes which utilize the architectural characteristics of a concurrent system (interconnection topology, processing power, and memory system).

Second, the design objective as an interactive image processing executive was to provide an easy-to-use image processing environment which provides a user a wider range of expressional tools and choices of interaction levels. The expressional tools allow a user to carry out tasks efficiently. The multiple choices of a user interaction level, according to experience level and application needs, allow a user to be acquainted with CIPE easily while providing room to grow with it.

Third, the design objective as a high rate data processing executive was to manage data efficiently so that its processing can be performed without wasteful file transactions or data traffic between the host and a concurrent system. The main goal of the data management scheme was to overcome the data manipulation difficulties in a concurrent system with localized memory and a serious data transfer bottleneck.

Finally, the overall design objective of CIPE is to provide an architecture-independent environment where neither a user nor a programmer needs to be involved in the system-dependent details related to the file system, display devices, or concurrent systems. Such an environment is pursued by separating the executive into device-dependent and independent modules and coupling them with a set of generalized interface routines. The decoupling of the architecture dependency from the functionalities allows CIPE to be easily upgraded to future system configurations as well as maintain the transportability of the application programs.

1.3. DESIGN IMPLEMENTATION

The current implementation of CIPE uses the combination of a MASSCOMP 5600 host computer and a MARK IIIfp 8-node hypercube (Figure 1.1). CIPE consists of four modules: user interface, host system monitor, hypercube monitor, and a set of generic image processing functions. Each module is designed and implemented to achieve the overall design objectives of CIPE.

As a user environment, CIPE provides a friendly user interface in three different forms: a command line interface, a menu interface, and a graphical windowing interface. The command line interface provides a simple interpreted programming language allowing interactive function definition and evaluation of algebraic expressions. The menu interface employs a hierarchical screen-oriented menu system for executing CIPE commands. The windowing interface is for graphical expression of commands and program flow. In this mode, the user performs functions through interactive creation of a graphical flow chart. Any of these interfaces can be used to control image processing operations under CIPE.

The host system monitor handles overall system interactions and provides generic image processing executive functions. The system management involves peripheral device handling, the concurrent system interface, and application

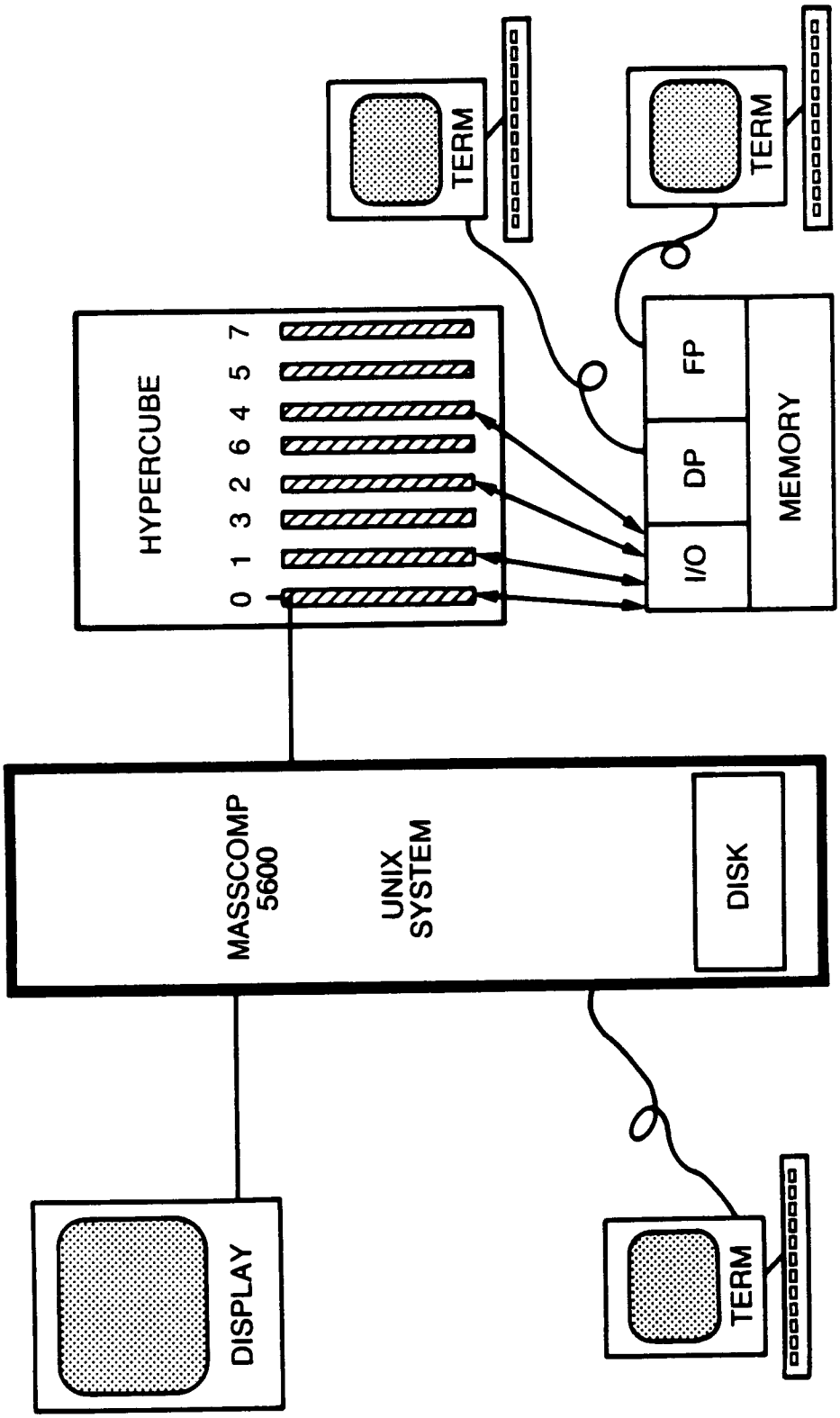

Figure 1.1 Concurrent Image Processing System Configuration

program execution. In order to develop an image processing executive for a virtual concurrent system environment, system setup procedures, data management schemes among multiple systems, and concurrent system interface methods were designed and implemented. Also, a data representation method was devised to optimize the large volume data manipulation among file system, host system, and concurrent systems.

Each concurrent system resident executive must implement unique architecture-dependent data and program management methods for optimal utilization of the architecture. The *hypercube resident monitor* allows CIPE to avoid data communication bottlenecks by keeping active data resident in the hypercube and minimizing application program space by downloading code as required to operate on the data. In order for data to be shared among several application programs with different data distribution requirements, an automatic data redistribution method is devised.

The image processing function module is a non-resident component of CIPE, since each function is dynamically loaded only for execution. The executive-provided functions (partly implemented) include generic image processing functions, systematic flight data processing, multi-spectral data processing, and a set of data processing primitives for interactive algorithm development.

The overall relation among modules is displayed in Figure 1.2. The example concurrent system resident executive in Figure 1.2 is a hypercube resident executive. The existing software tools employed for CIPE development are virtual display interface software developed by MIPL/JPL, and a general purpose menu manager software developed by Alan Mazer. CIPE is written in the C Programming Language and utilizes Yet Another Compiler Compiler (YACC) for command line parsing, and SunTool for the windowing user interface. The hypercube resident software is developed under the Crystalline Operating System (CrOS) supported by Section 335/JPL using the C Programming Language.

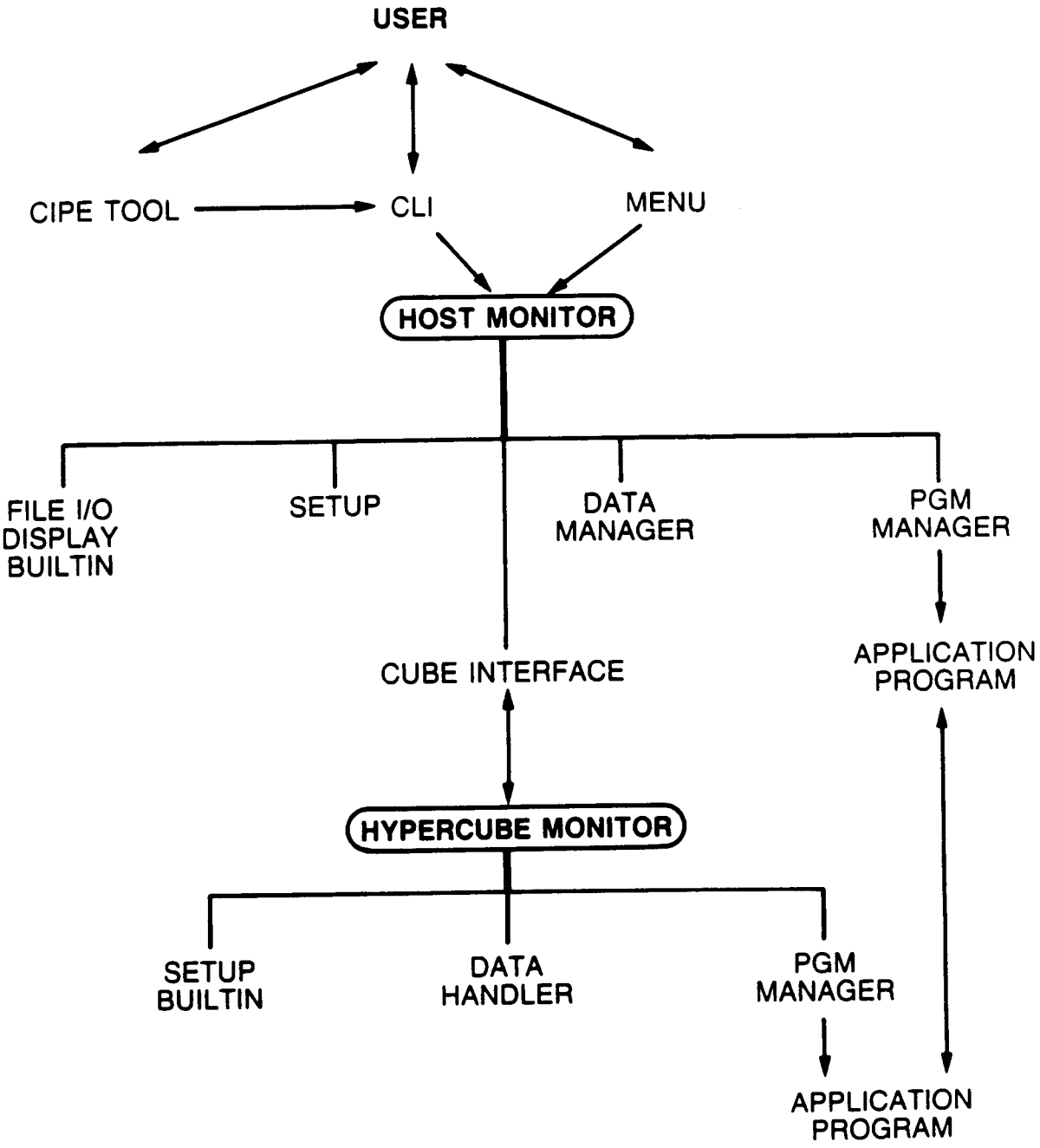

Figure 1.2 CIPE Software Structure

2. USER INTERFACE

The design philosophy behind the CIPE user environment is to provide the user a choice of interfaces. Though numerous "user friendly" interfaces have been developed, there is no single interface scheme that all users agree on; individual users have different conceptions of "friendliness" just as they have different levels of expertise and different processing needs.

Basically, there are three types of user interface modes: command line, menu, and windowing. The command line interface (used by computer operating systems, for example) imposes a rigid syntax structure and usually demands complete information about the operation to be performed when the command is entered. It is especially useful in a batch-oriented environment where a set of precomposed command lines can be utilized repeatedly, or when functions and expressions need to be interactively defined and evaluated. The menu interface offers a hierarchical organization and preformatted prompts for necessary information which a user can easily follow in real time. Its usage is for interactive data processing environments with limited amounts of user-provided information. The windowing interface provides another dimension to user interaction by offering a graphical expression. Its usage is for visually oriented data processing environments where a simple graphical expression can replace a complicated command line or menu input.

CIPE provides all three types of user interface, allowing the user the advantages of each. The command line interface (Cli) provides a simple interpreted programming language allowing interactive function definition and evaluation of algebraic expressions. The menu interface employs a hierarchical screen-oriented menu system. The windowing interface is for graphical expression of commands and program flow. In this mode, the user performs functions through interactive creation of a graphical flowchart. Any of these interfaces can be used to control image processing operations under CIPE.

The windowing interface is implemented as a separate process which passes the user request to CIPE after the user expression is converted into an equivalent command line. This scheme was chosen to minimize the work of checking for valid user requests and to develop an executive-independent user interface.

Cli and menu mode implement user input error-checking for all provided applications to offer rapid response to input errors. Performing error checks at input time reduces the probability of execution error and prevents wasteful execution.

In this section, the discussion of each user interface mode is focused on describing CIPE's usage of each mode with respect to user interface organization and functionality through descriptions and examples. Details on the menu manager (Yamm) and windowing interface (Cipetool) are included in appendices.

2.1. COMMAND LINE INTERPRETER

The first of the user interface approaches implemented in CIPE is the Command Line Interpreter (Cli). The Cli makes available the basic capabilities of CIPE, including environment control, program execution, and the evaluation of functions and expressions. It also provides for the definition of Cli-interpreted functions and scripts, allows the creation of *workspace* collections of user-defined procedures, and provides more terminal independence than the other user interfaces. Interpreted functions may be expressed in terms of built-in primitives, compiled code, or other interpreted functions. The command line approach allows operations to be performed repetitively through looping and simplifies interactive evaluation of complex expressions using built-in and user-defined functions. Since most operating systems are command line based, the Cli approach also allows a user to escape CIPE execution temporarily and run operating system utilities such as to list a directory or to edit a file.

This section discusses the implementation and use of the Cli, which has four parts: command line input and lexical analysis, parsing, code generation, and execution. Commands are taken from the keyboard or workspace, separated and unaliased into tokens in the command language, and parsed; commands taken from the keyboard may have been edited. The parser uses syntax-directed translation to build a pseudo-code program, which is then executed. The following subsections discuss each of these functions in detail. The section concludes with an annotated example showing how the Cli works in a typical user session, and a brief discussion of the application program interface.

2.1.1. Input and Lexical Analysis

The input and lexical analysis phases of command line interpretation are straightforward, centered around the concept of *input streams*. An input stream corresponds to a source of command input. As new streams are added, descriptors for previous streams are pushed onto a stack until such time as the new stream runs out of input. The first input stream in any session is the *standard input*, by default, the keyboard. As a user typing at the keyboard loads a previously saved workspace, the workspace file becomes the current stream and the descriptor for the previous stream is pushed onto the stack. Workspace execution continues until the workspace loads another workspace or ends, at which time the stack is adjusted accordingly. Edited command lines are handled similarly, written to a file which is then opened as a stream and afterwards discarded.

Command line interpretation starts with initializations: the allocation of initial buffer space, and the creation of an initial input stream descriptor. The remainder of interpretation is then a loop in which input handling is simply determining the current stream, managing pointers and status variables, providing for editing, and reading a line. If the user requests editing of a previous or current command using CTRL E, the input code writes out what has been typed

so far, or if nothing, the last command, and allows the user to edit the command using a standard full screen editor. This approach was chosen over more simple, line-based command editors both because of the ease with which a user can access a favorite editor and because CIPE command lines, such as those containing function definitions, are potentially so complex that a line editor approach is very difficult to use. Once edited to the user's satisfaction, the descriptor for the previous stream is temporarily pushed onto the stack, and input is taken from a temporary file containing the user's edited command. Implicit in the process of command input are management requirements such as the monitoring of buffer space (which may be automatically increased if necessary), the removal of unnecessary space and comments, and for edited command lines, feedback of the final command line to the user.

Actual lexical analysis is done by the function, *yylex*, generated from a legal-token specification by the Unix* utility *lex*. Supporting this lexical analysis are a variety of functions within Cli for tracing, line continuation handling, input buffer access, recognition of reserved words and their types, and error handling. Line continuations are usually provided automatically. Whenever a user enters a command line which is incomplete, either because some terminating delimiter has not been entered (such as a *for* loop started but not finished with an *end* or a function call with an incomplete argument list), or because the line ended with a two-argument operator, the Cli prints a continuation prompt, properly indented, to show that more input is expected. Alternatively, a user can indicate that more is to come by ending an input line with a backslash(\). Reserved words are recognized using a variety of lists stored within the lexical analyzer. Because reserved words such as multiword system attributes may be specified in several ways (with underscores between words of the attribute, or spaces, and with component words abbreviated), their recognition requires some local parsing to determine whether they are reserved words, legal variable names or simply errors. In addition to system attributes, reserved words include commands, command support words (e.g., *step* in *for i=0 to 10 step 5*), and boolean and keyword values (e.g., *on*). While error detection is largely done by the parsing, alerting the user is a function of lexical analysis, largely because the lexical analyzer has more information about the actual command lines as entered than does the parser. On errors, the lexical analyzer either prints the command line causing the error, together with an explicit error message and a carat (^) pointing to the location of the error within the line, or displays a workspace name and the number of the line within the workspace causing the error.

2.1.2. Parsing

The tokens taken from the input stream by the lexical analyzer are next fed into the parser, which attempts to assemble the tokens into a legal form, while at the

* Unix is a registered trademark of AT&T Bell Laboratories.

same time generating pseudo-code for the eventual execution of the command. The parser itself is generated by the Unix utility *yacc* from the Cli grammar definition file *parser.y*.

The *yacc* language definition consists of two parts: productions and parse actions. The productions are defined formally by the following grammar in Backus-Naur Form:

```
    cmd_list ← cmd_list command
             | command

   command ← simple_cmd
             | cntl_struct

simple_cmd ← exec_cmd
             | ! unix_cmd
             | output = expr
             | func_name ( actual_args )
             | func_name
             | quit

 cntl_struct ← for var_name = expr to expr scmd_list end
             | for var_name = expr to expr step expr scmd_list end
             | while expr scmd_list end
             | if expr then scmd_list endif
             | if expr then scmd_list else scmd_list endif
             | if expr then scmd_list else_if_list else_list endif

  exec_cmd ← set entity attr to expr
             | set attr to expr
             | turn bool entity attr
             | turn bool attr
             | functions
             | symbols
             | traces
             | print expr_list
             | show func_name
             | define func_name ( formal_args ) cmd_list end
             | load ws_name
             | save ws_name
             | edit ws_name
             | read var_name from filename
             | write var_name to filename
             | display var_name at_loc
             | menu
             | nomenu
```

```
                | quit

      output ← var_name
                | var_name [ index_list ]

    scmd_list ← scmd_list ; simple_cmd
                | simple_cmd

  else_if_list ← else_if_list else if expr then scmd_list
                | else if expr then scmd_list

     else_list ← else scmd_list
                | ε

       entity ← cube | gapp | system

         attr ← dimension | num_procs | topology | priority | logging
                | lextrace | parsetrace | codegentrace
                | exectrace | symtabtrace | functabtrace

  formal_args ← ε
                | var_name
                | formal_args , var_name

     ws_name ← expr

     filename ← expr

       at_loc ← ε
                | at expr , expr

   index_list ← index_list , index
                | index

  actual_args ← ε
                | expr_list

        index ← expr
                | expr : expr

     expr_list ← expr
                | expr_list , expr

          expr ← expr op2 expr
                | op1 expr
                | var_name
                | var_name [ index_list ]
```

```
                | func_name ( actual_args )
                | ( expr )
                | constant

      constant ← single_value
                | { value_list }

    value_list ← value_list , single_value
                | single_value

  single_value ← integer
                | float
                | string
                | enum

          op2 ← &&  |  ||  |  &  |  |  |  ^  |  <  |  <=
              |  ==  |  !=  |  >=  |  >  |  *  |  /  |  +  |  -

          op1 ← -  |  !  |  ~
```

A few formats and details are omitted here where the rules are generally agreed upon, such as for the formation of *floats*. Variables are not explicitly typed; types are determined at run-time and functions do type conversion as necessary.

Yacc's output code uses *shift-reduce* parsing with single-token lookahead to reduce every input line to a valid command. As tokens are received from the lexical analyzer and *shifted* onto a stack, the parser attempts to *reduce* (combine) sequences of tokens appearing on the right side of one of the listed productions. When several tokens can be reduced to a higher-level element of the grammar, called a *non-terminal* to distinguish it from a *terminal* or reserved word in the command language, they are replaced on the stack by the non-terminal, which is in turn compared against the right side of the productions above. Ultimately, the group of tokens forming the input line reduce, through many intermediate steps, to a *cmd_list*. Note that a legal input line may actually have multiple commands according to this grammar. No command delimiter is necessary; the grammar is designed such that successive commands may be expressed on the same line unambiguously.

In addition to the productions listed above, the yacc input definition contains parse actions. Parse actions are invoked whenever a sequence of tokens or non-terminals reduce to a higher-level non-terminal, and represent most of the code in the yacc input definition. They fulfill two functions: to assist in parsing, and to create the pseudo-code which is the ultimate product of the parser. For example, when an integer value is sent by the lexical analyzer, the parser recognizes that and reduces it to the non-terminal *single_value*. The parse

action for the reduction is to print out a trace, if the user has requested parse tracing, allocate storage and save the value in the symbol table, and then associate the index of the value in the symbol table with the new non-terminal on the parse stack. Similarly, the non-terminal single_value will eventually be used along with other input in the formation of one of the non-terminals *constant* or *value_list*, depending on whether or not it was preceded by a brace ({). If part of the non-terminal value_list, the parse action combines the value represented by single_value with the other values in the value list, associating with the new non-terminal this expanded list of values. Ultimately, the instruction using this value will be generated by yet another parse action looking at a correspondingly higher-level group of non-terminals.

These and similar parse actions are applied at every step in the parsing of the grammar until a program has been created. Other parse actions handle such things as intelligent error detection and recovery, table lookups and management, backpatching (changing previously created code based on new and previously unavailable information), code segment management (described in more detail in the next section), and the creation of temporary variables.

2.1.3. Code Generation

A large part of parser action work is in the generation of pseudo-code instructions for execution of the command. The code called by these actions forms the third part of the Cli, code generation. As the idea of streams is fundamental to understanding command input and lexical analysis, so *code segments* are central to code generation. A code segment is a data structure containing the name of a user-defined function, a pointer to the compiled code, a symbol table and function table, and a pointer to a *parent* code segment. One code segment is set up by default when the user starts up CIPE. Commands typed at the keyboard are compiled and described by this code segment, including any functions defined at the keyboard level and any symbols created interactively. When a workspace is loaded, a descriptor is created for it, with the original code segment as parent, and execution shifts over to the workspace. The use of code segments allows easy grouping of symbols and subordinate functions in the proper context, and allows CIPE to quickly find the correct data or function when a specification is potentially ambiguous.

The code generation functions fall into two groups: code segment maintenance and support, and instruction generation. Code segment maintenance routines include code to create new code segments (as when executing a DEFINE command), add instructions to the current code segment, and return to a previous code segment. Support routines store and manage constants in the symbol table, perform compile-time type conversions, disassemble instructions for user-defined workspace and function listings, and create temporary variables. Instruction generation routines take information grouped by the parser and translate it into individual instructions. Most commands recognized by the parser translate into a specific instruction which is generated by a corresponding

code generation routine. For example, the routine *cipe_arith_op* is called when the parser gets a token group of the form *constant + constant*. It generates an addition instruction, creating a temporary variable, if necessary. Similar code generation routines create instructions for assignments, subscripting, function calls, and the other language features. In the case of branching as part of a looping construct, where all the available information for instruction generation may not be available until the loop is completely parsed, parser actions direct the routines to produce skeletal code which is then backpatched, or filled in as the information becomes available. Code generation for the assignment instruction attempts to modify previous code to make the assignment instruction and the resulting overhead unnecessary.

2.1.4. Execution

The fourth and final step in command line interpretation is taking the instructions created by the code generation routines and executing them. At this point, the lines between the user interface and the core of CIPE begin to blur. Execution of the generated instructions is distinctly a Cli responsibility, but this responsibility is implemented using basic CIPE system functions. Not all code generated is executed immediately either. If a user types in a function definition using the DEFINE command, two pieces of code are generated. The first of these is the code for the function being defined. The second is a very short program *defining* the function as a function in the current code segment. This definition basically associates the name of the function with the location of its code segment in the current function table. The function defined, although it has been input, parsed, and translated into pseudo-code, is not executed until a call to the function takes place. Only the definition instruction is actually executed immediately. This distinction is very important.

When code is executed, the process is straightforward. Execution is implemented as a loop where each iteration grabs an instruction from the current code segment, displays it if requested by the user using the *TURN ON EXEC TRACE* command, and then executes it. Generally, control will then pass to the next instruction which will be executed in the next iteration of the loop. Branches and QUIT instructions may be used to alter the normal flow. All instructions do run-time error checking and type conversions as necessary. Arithmetic built-ins are executed on the coprocessor if the data size is sufficiently large; otherwise they are executed by the host system. When a call to a user-defined function is executed, the function is located in the current or an ancestor code segment's function table and the current code segment is replaced with the code segment of the function being called. When execution of that function finishes, possibly after calling other functions, the original section of code resumes execution. Recursion is not permitted. Compiled functions may also be called, but do not result in any changes in the current code segment.

2.1.5. User Summary and Example

Subsections 2.1.1 through 2.1.4 have described in some detail the major parts of command line interpretation, that is, input and lexical analysis, parsing, code generation, and the eventual execution. This section presents a more informal description of the available commands and an example showing their usage in a typical terminal session. The commands are as follows:

set attr **to** expr
turn bool attr

> *Attr* is an attribute (e.g., **coprocessor, display device, logging**). An expression is formed using (possibly subscripted) variable names, function calls, operators, and constants (strings, numerics, keywords). C language operators work with the same precedence as in C. Constants may be multivalued if values are separated with commas and the constant is delimited by left and right braces. *Bool* is a boolean value (i.e., **on, off**).

functions
symbols

> Display the current functions and symbols.

traces

> Show the traceable system parameters and their trace status.

print expr_list

> Display the specified value(s). Multiple values should be separated with commas. A newline is added automatically.

show func_name

> List the specified user-defined function in a pretty format.

define func_name (formal_args) cmd_list **end**

> Define a new function.

load ws_name

> A *workspace* is a set of commands or functions definitions or both. They are stored as text on the disk. A *ws_name* is a quoted file specification. *Load* retrieves a workspace from the disk. Multiple workspaces may be loaded at the same time.

save ws_name

Save all current user-defined functions to the disk.

edit ws_name

Edit the specified workspace using the *vi* full screen editor.

read var_name **from** filename
write var_name **to** filename

Read or write using specified variable and file *filename*.

display var_name
display var_name **at** line , sample

Display image on current display device at top-left of screen or position *line*, *sample*.

menu
nomenu

Enable and disable menuing. If in a workspace file, these commands do not take effect until execution of the entire file has completed.

quit

Exit CIPE. Control-D works also.

! unix_cmd

Performs the specified Unix command.

output = expr

Assign the value of *expr* to variable or subscripted variable *output*.

func_name (expr_list)
func_name

Call function *func_name*.

for var_name = expr **to** expr cmd_list **end**
for var_name = expr **to** expr **step** expr cmd_list **end**

while expr cmd_list **end**

if expr **then** cmd_list **endif**
if expr **then** cmd_list **else** cmd_list **endif**
if expr **then** cmd_list **else if** expr **then** cmd_list... **else** cmd_list **endif**
if expr **then** cmd_list **else if** expr **then** cmd_list... **endif**

Cmd_list is a list of commands from Section 2 above; these are easiest to read on separate lines, but they can be run together if desired. Control structures are not nestable at the moment.

The following is an example of a terminal session. A pound sign (#) indicates the start of a comment. These comments need not be typed in as part of the command lines, but are useful in complex user-defined functions. All text following a pound sign is ignored by CIPE. User input is underlined.

```
unix% cipe
CIPE Version 3.2                    # local startup script performs
                                    # user-specific initializations on startup

> set coprocessor to cube           # grab the hypercube
> turn on logging                   # start session logging – this could
                                    # have been done in startup script
                                    # automatically
> set disp dev to ivas              # allocate ivas – note abbreviations
> read aisa from "/ufs/images/aisa" # load an image

> filtered = boxfilter(aisa,3,3)    # programmer-defined function returns result

> stretched = stretch(filtered)
> display stretched                 # keywords for output include print, write,
                                    # and display

# At prompt, define general procedure to do the steps above

> define filtnstretch(image,filt_nl,filt_ns)
?    load image
?    boxfilter(image,filt_nl,filt_ns)  # do filter in place here
?    stretch(image)
?    display image
?    end
> ^E                                # user decided to add to previous
                                    # command line
```

```
... vi ... vi ... vi ...

# on exit from vi, system executes new command

> define filtnstretch(image,filt_nl,filt_ns)
?    if filt_nl < 1 then
?       print "Number of lines (",filt_nl,") is too small"
?       filt_nl = 1
?       endif
?    if filt_ns < 1 then
?       print "Number of samples (",filt_ns,") is too small"
?       filt_ns = 1
?       endif
?    boxfilter(image,filt_nl,filt_ns)
?    stretch(image)
?    line=1
?    for sample=1 to 511 step 32
?       display image at line,sample
?       end
?    print "The new image is displayed now!"
?    end
>

> filtnstretch("/ufs/images/aisa",3,3) # user tests function
The new image is displayed now!

> save "myfunctions"                   # user saves function on disk
                                       # for future use
> !ls -l myfunctions                   # check directory for saved
                                       # workspace
-rw-r--r--  1 cipe       4322 Jul  7 13:05 myfunctions
> quit
```

2.1.6. Application Program Interface

The Cli application program interface is extremely simple. When a compiled application function is called, Cli parses the command line, checks the function dictionary to find the location of the code, and invokes the function. Descriptors of the arguments passed to the function by the user are stored internally. The application program accesses these descriptors as an external array and calls CIPE system routines to extract the values of the arguments passed on the command line. When an argument is an expression rather than a symbol, CIPE evaluates the expression and creates a temporary symbol to use in the argument list. If the function has a return value, the symbol designated by the user for the result is also referenced as an external and used by the application programmer as storage for the final result.

2.2. MENU

The menu interface of CIPE serves several purposes: self-orientation to CIPE's organization, a simple environment for inexperienced users, interactive input verification, and functional grouping of CIPE capabilities. The entire CIPE functionality is accessible by flipping through menus using menu control keys, allowing one to quickly grasp the organization and usage of individual CIPE commands. Each activated command displays prompts for required user inputs and help can be made available for each input field so that inexperienced users can easily follow the procedure. The validity of input may also be checked in real-time and proper error messages displayed so that users can correct input before actual execution takes place. The hierarchical organization of the menus allows grouping of related functions. This functional grouping enhances the menu entry selection process, since a user can scope CIPE according to his/her processing needs.

Menu mode is implemented using Yamm (Yet Another Menu Manager), a general purpose menuing package. Design details of Yamm are included in Appendix G and an implementation example is provided in Section 5. This section is devoted to describing the utilization of the package in CIPE to provide a menuing user interface to serve the purposes presented above.

2.2.1. Menu Format

Yamm employs three windows: a menu window, an application I/O and data entry window, and a status window. The menu window displays the submenus and functions available at the current level of the menu tree. The menu hierarchy is specifiable on a per-user basis using *menu configuration* files, allowing individual users to customize the environments. The menu configuration is structured in a multibranched tree fashion, where the leaf node is connected to a corresponding routine name as shown in Figure 2.1.

Currently, accessible submenus and applications are selected either by number or by using arrow keys to move to the desired selection. If a submenu is selected, its set of menus and accessible functions replaces those of the previous menu in the menu window. When the selected submenu entry is a leaf node in the menu configuration, the corresponding function becomes activated. The interaction between a user and the activated function takes place in the application I/O and data entry window.

The application I/O and data entry window serves a dual purpose. During execution of user programs, it displays application output and system error messages. During application-requested data entry, the window displays a data entry form for specification of a given command's inputs. A parameter may be classified as required when a value must be given by a user or defaulted. For the parameters with default values, the default values are automatically displayed and a user may accept or change them. Individual parameters may

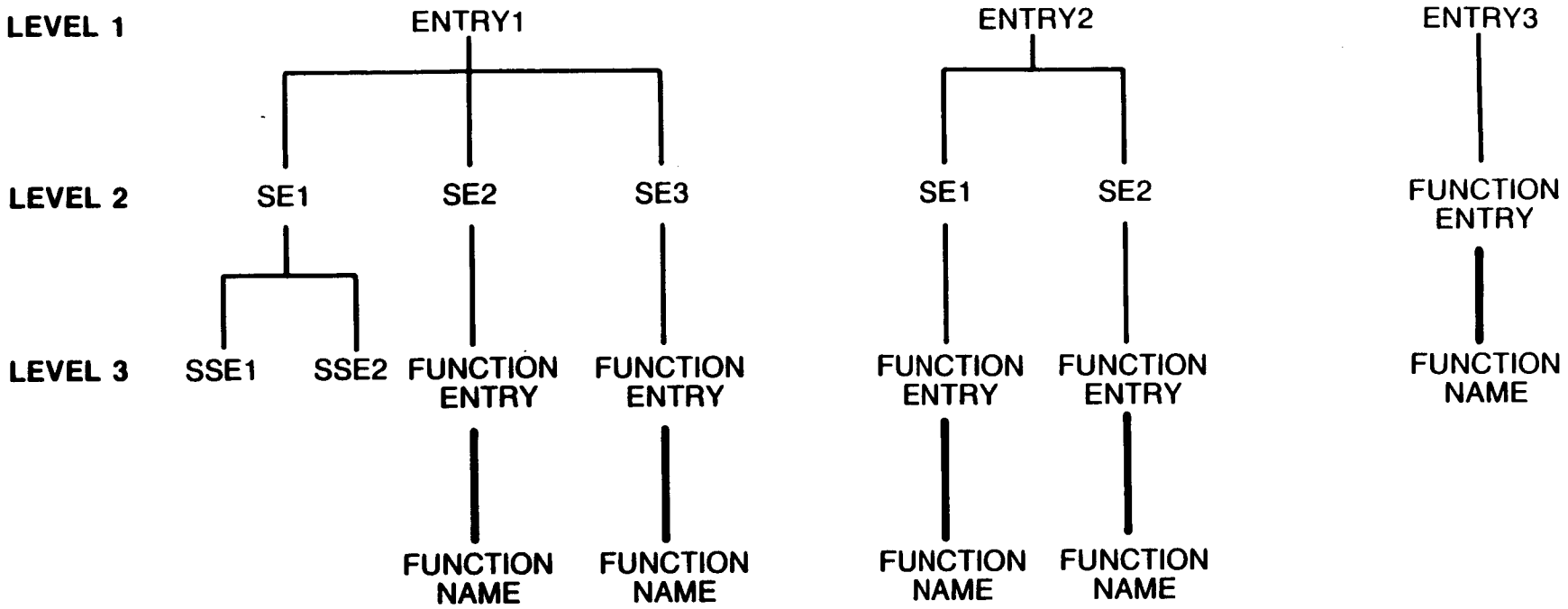

Figure 2.1 Menu Structure

also have help and parameter-specific error checking. For parameters with just a fixed number of valid values, the key combination shown in the status window next to **Next Value** may be used to step through the possibilities. See Appendix G for the complete set of Yamm capabilities.

The status window displays the name of the current menu and the keys used for the available functions in context. For example, during parameter entry the window shows keyboard mappings for **Return to Menu, Help, End Data Entry**, etc. When waiting for a submenu or application selection, the menu status window shows mappings for such things as **Previous Menu** and **Exit**. Keyboard mappings are determined automatically based on the type of terminal in use, but may also be specified in the code by the programmer or in user-specific menu configuration files.

2.2.2. CIPE Menu Configuration

The CIPE menu hierarchy is structured so that a user can easily understand and access the available functions. The hierarchy has two major components: CIPE commands and application functions (Figure 2.2). The first group includes entries for coprocessor-independent functions such as help, system setup, symbol management, display, and plot utilities. The application functions menu will be modified as different application functions are developed. The current applications menu includes a set of generic image processing capabilities. The CIPE menu configuration can be found in Appendix I under *menuconfig* and the link between the function entries and corresponding function names are defined in *menus.h*, also in Appendix I.

The symbol management entry is a submenu with a set of lower level entries including **List, Assign, Read, Save** and **Delete**. The **List** entry is for listing out the the currently defined symbols. The **Assign** entry allows the user to assign a set of values to a symbol. The **Read** entry associates an existing file with a symbol. The **Save** entry is for saving symbol values to files. **Delete** deletes specified symbols from the system. Figure 2.3 illustrates the symbol manipulation menu where the **Read** function is activated. Actual execution of these functions is discussed in Section Three under Data Management (Section 3.2).

Display capabilities include display device allocation, stretches, image display, histograms, and display control. Display devices may be allocated in black and white, color, or pseudo-color modes. The **Stretch** entry allows linear and non-linear lookup table manipulations. Both two- and three-dimensional image data may be displayed and used in histograms. Display control capabilities include cursor access and control, and erase and zoom operations. Figure 2.4 shows the display menu where **Draw** function is activated. Plot capabilities include multi-spectral plotting and three-dimensional image plotting where pixel intensity is represented as height.

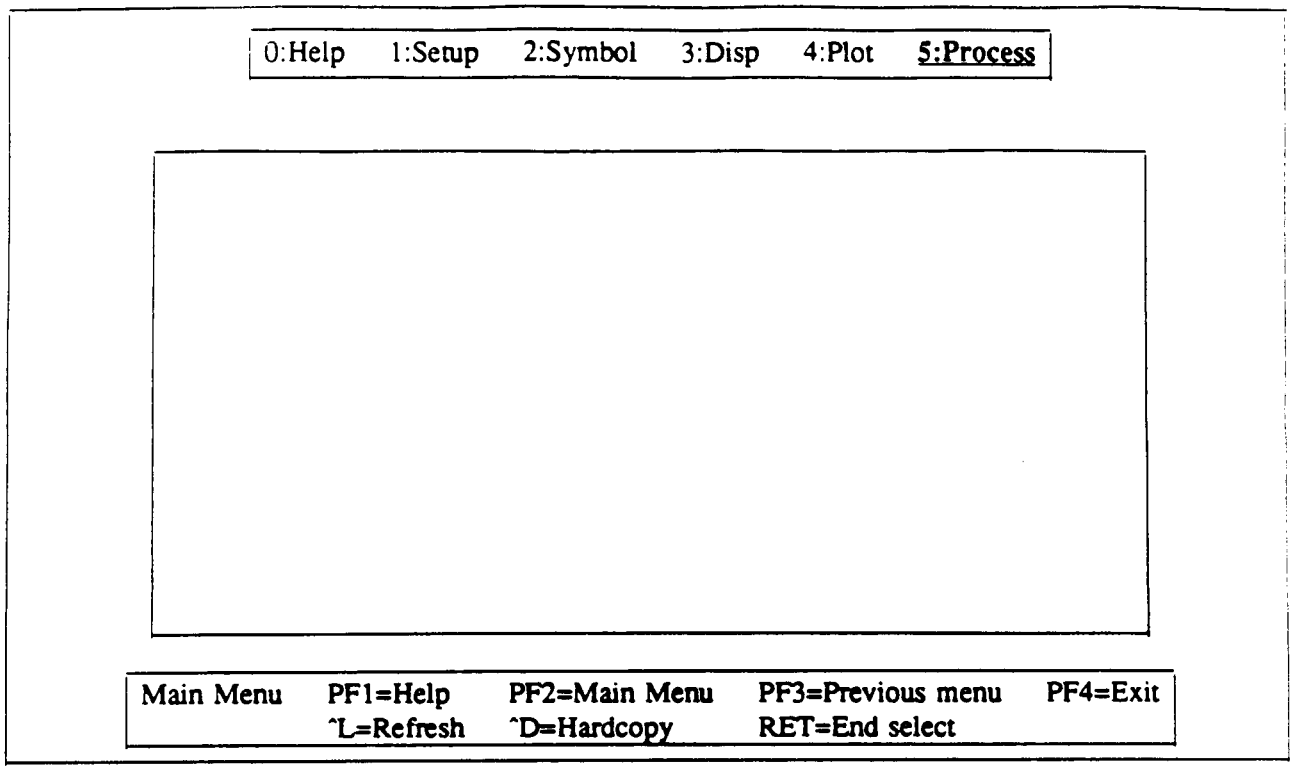

Figure 2.2 Root Menu of CIPE

Image processing applications functions are subdivided into **Enhance** (Image Enhancement), **Geom** (Geometric Transformation), **Classify** (Scene Classification), **Builtin** (Built-in Functions), and **Appl** (User Applications). Image enhancement capabilities include spatial convolution, histogram manipulation, frequency domain filtering (low/high/band pass) and statistical (median) filtering. The geometric operations include tie-point-oriented geometric correction, map projection, and image resizing. The classification functions include textural scene segmentation and spectral classification. The user application entry activates user-developed application programs compiled outside of CIPE. Details on writing application programs for CIPE are discussed in Section 5, Application Programming in CIPE.

Along with these other capabilities, CIPE provides a set of built-in functions for arithmetic, relational, and logical operations on scalar and multi-dimensional data. Each of these takes one or more symbols as arguments and returns a symbolic result. Figure 2.5 shows the image processing menu where the **Builtin** function is activated.

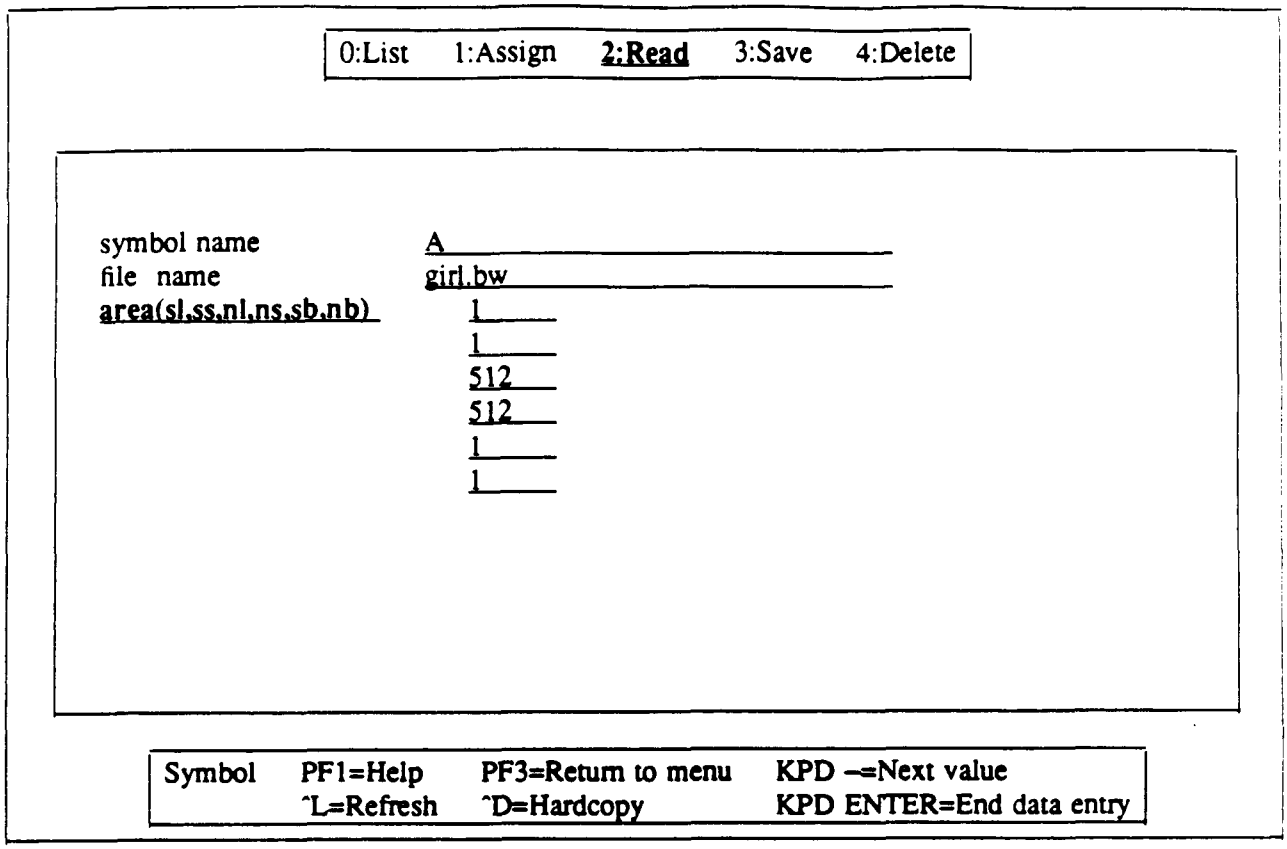

Figure 2.3 Symbol Menu

2.2.3. Parameter Field Structure

Apart from determining a menu hierarchy, the other major task in implementing the menu interface is the design of individual data entry forms to be displayed by Yamm when specific CIPE commands or application functions are requested. For each input parameter Yamm needs to know such things as a screen location, prompt message, data type (for Yamm internal error checking and conversion), and information about number of values, default values, and field widths. CIPE provides this information to Yamm for each parameter field and Yamm then uses these in displaying the data entry form and processing input. An example parameter specification internal to CIPE and its Yamm interpretation follows. Function-dependent error checking and interactive help are also available and they are discussed in the next two sections.

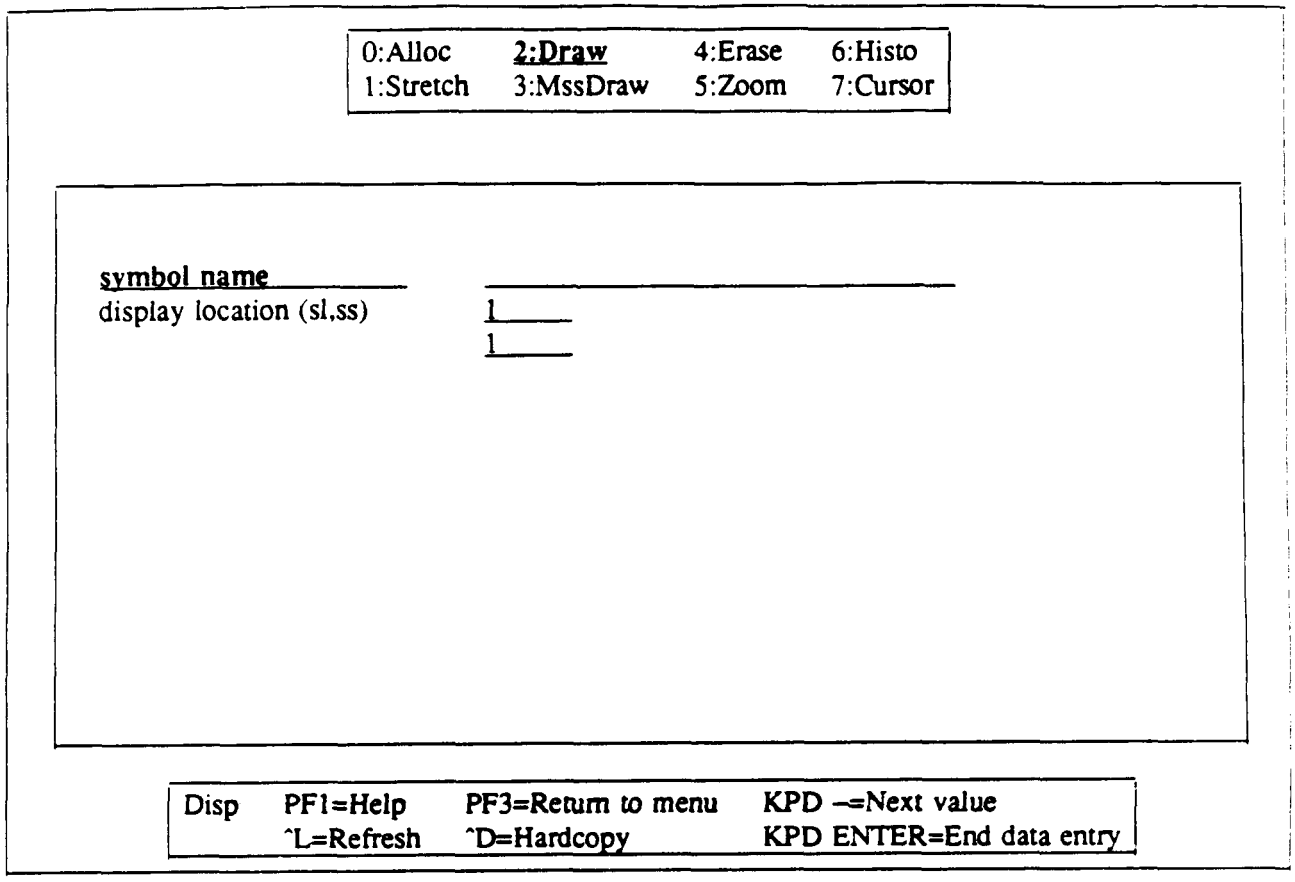

Figure 2.4 Display Menu

```
param_count = 4;
def(params,0,"input symbol name      ",STRING,input,WID,32,
   LINE,2,REQ,END);
def(params,1,"outut symbol name      ",STRING,output,WID,32,
   LINE,3,REQ,END);
def(params,2,"filter window(nlw,nsw) ",INT,window,INCR,4,
   WID,10,LINE,4,DUP,2,REQ,END);
if (getpar(params,param_count,check_filter,help_filter)) return;
```

When the statements in the example are executed by Yamm, a menu shown in Figure 2.6 will be displayed in the terminal screen.

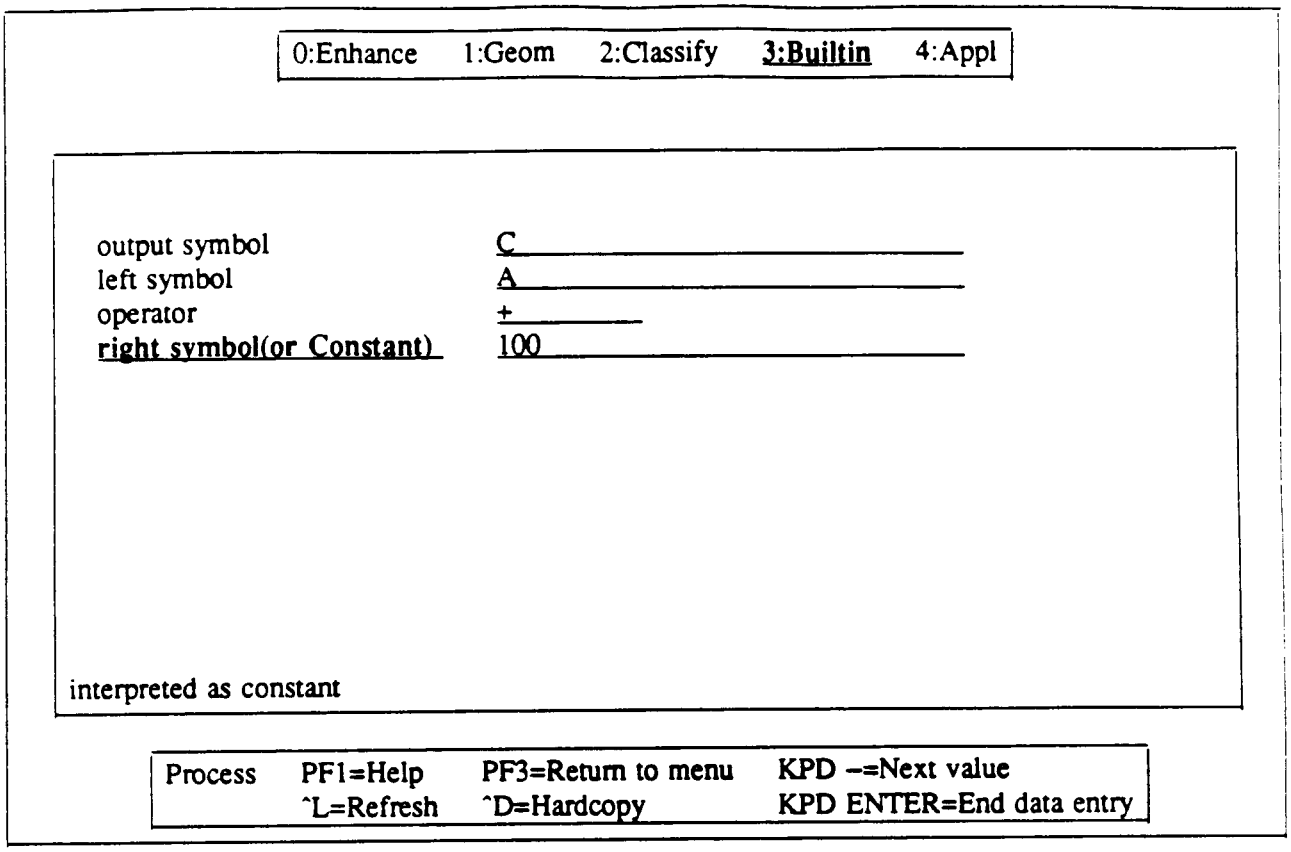

Figure 2.5 Image Processing Menu

2.2.4. Error Handling

Another part of parameter entry definition is the specification of an error checking function by which input validity may be checked as input is entered. This interactive user input verification significantly enhances interactive use of CIPE because it prevents wasteful execution of invalid input parameters. The error checking mechanism is provided by Yamm and the actual value checking is provided by CIPE. Yamm's input processing routine (**getpar**) allows optional specification of an error checking routine. When an error checking routine is provided, Yamm calls the specified routine for each parameter input value. The correct error checking steps are then selected based on the parameter number.

CIPE provides an error checking routine for Yamm to call for each function. Each routine is composed of several sets of error checking steps, each set covering a specific input field. The error checking steps evaluate the input value and print a proper error message if the value is found to be invalid for the activated function. Since syntax errors are checked by Yamm, only validity

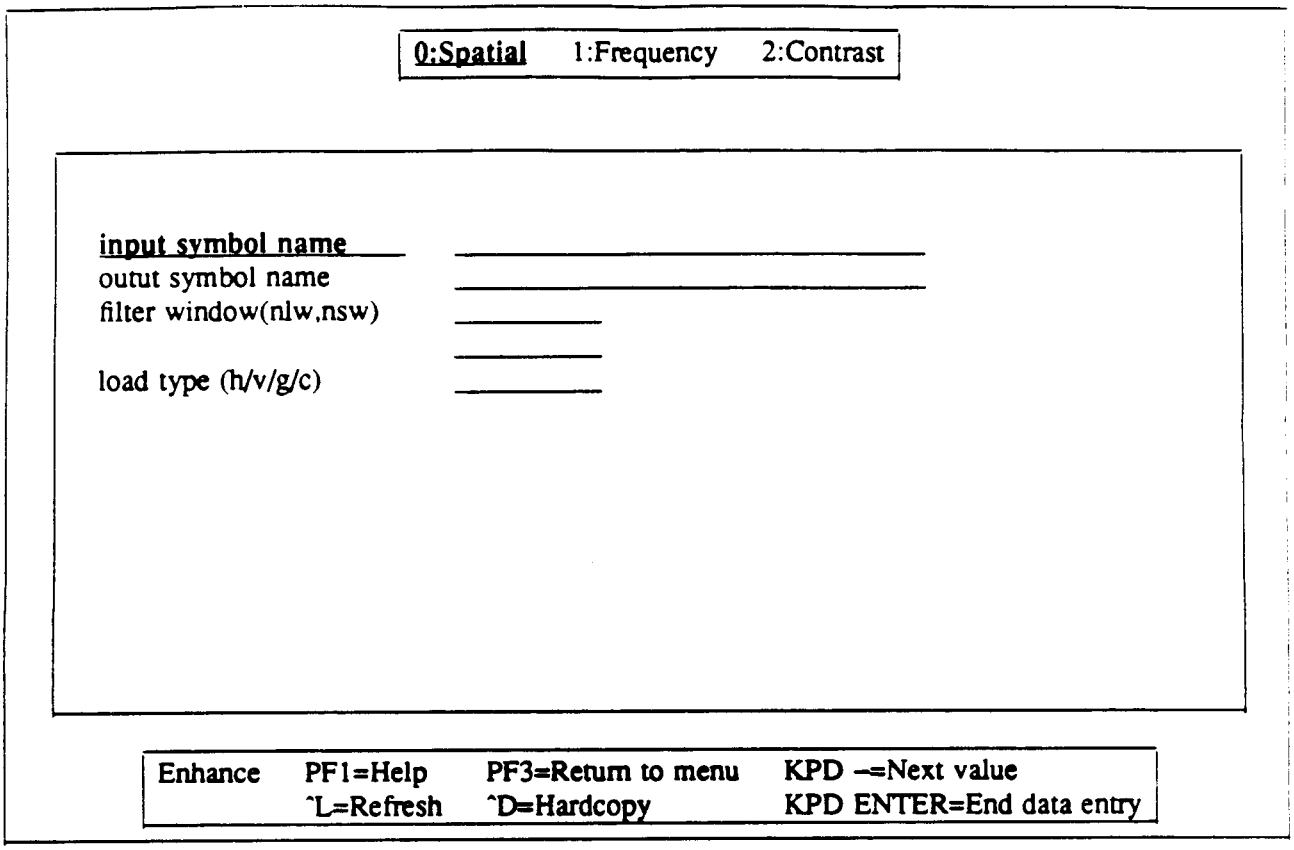

Figure 2.6 Spatial Filter Parameter Menu

checking of entered values is necessary. Yamm uses the last two lines of the application I/O and data entry window for the error/warning messages.

As shown in the example of the error checking routine for a spatial filter program, error conditions are handled for each parameter using a case statement. In case 1, the input symbol name is checked for existence and a message is printed on error. In case 2, the output symbol is checked for non-existence and a warning message printed if an existing symbol may be overwritten. In case 3, the window parameter is verified for a valid range as well as an odd value. An illegal value is replaced with the previous valid value or default value (if any) when the error message is displayed.

```
check_filter(params,number,value)
struct param *params;
int number;
union {int i; double f; char *s;} value;
{
        int error,whereis;
        error = 0;
        switch(number){
        case 0: whereis=cipe_get_symbol_index(value.s);
                if (whereis == NO_SYMBOL) error = 1;
               if (error) error_print("symbol does not exist");
               break;
        case 1: whereis=cipe_get_symbol_index(value.s);
                if (whereis != NO_SYMBOL) error = 2;
                if (error) error_print("symbol already exists");
                break;
        case 2: if (((value.i) % 2) == 0) error = 3;
                if (error) error_print("must be odd number");
                break;
        }
        return(error);
}
```

2.2.5. Interactive Help Messages

Similar to the error checking mechanism, Yamm provides a help mechanism for individual input fields and CIPE provides the actual help messages. A help routine may be provided for each Yamm-invoked function; the routine name is passed to Yamm as a parameter of **getpar**. The help routines are structured and called similar to the error checking routines.

In order to display a help message, a user presses the HELP key as defined by the status window. Then Yamm calls the help routine to print the corresponding message. Help message examples include required value range, more detailed prompt messages, parameter usage, and various function-dependent guidelines.

The help routine for the spatial filter program is illustrated as an example. Some parameter fields (such as that for the file name) do not require help because they are very straightforward. Others (such as the filter window parameter field) provide an explanation of usage and a proper value range.

```
help_filter(params,number)
struct param *params;
int number;
{
        switch(number){
        case 0: printf("input data must be two dimensional image");
                break;
        case 1: break;
        case 2:
                printf("size of a weight matrix, must be odd number");
                break;
        }
}
```

2.3. CIPETOOL

Cipetool is a graphical user interface that serves as a front-end to CIPE. The freedom which allows a user to express his/her processing needs in the most direct fashion is highly correlated with the user's performance. The main goal of this user interface mode is to expand the user's expressional freedom beyond the limited keyboard typing of Cli and Menu mode of CIPE. Currently, a very limited conceptual level of the graphical interface has been implemented using iconic manipulation, based on the Sunview Windowing system supported by Sun Microsystem. This section describes the current status of Cipetool as an iconic interface mode to CIPE with respect to icon implementation and interaction with CIPE. The direction of the future Cipetool is also discussed.

2.3.1. Icons

Current Cipetool employs three types of icons: data, function, and device. The data type is subdivided into constants, arrays, and images. The constant and array icons are for entering immediate values to an activated function. A character string value is represented using a constant icon and it is indicated by surrounding it with double quotes. The image icons are associated with image files in the system and their sizes are made to be proportional to the actual image's size. An image icon can be manipulated to select a subsection of an image by specifying the starting and ending locations using a mouse. The function type includes all of the CIPE data processing functions. The function icons are associated with data processing functions of CIPE and their sizes are determined based on the function descriptions (prompt messages, numbers) of input and output parameters provided in the function dictionary. The input and output entries of each function are designated with prompt message boxes. A user must provide input parameter values by connecting proper input data icons to the prompt message boxes of a function icon before execution of the function. A prompt message box is referred to as a connection box since it is used for

connecting a data icon. Cipetool creates an output icon after the function is completed based on the data type (image, array, constant) and the size of the output. A display device is only device icon implemented at this stage. Thus, there are five icons, constant, array, image, function, and display device.

A user manipulates the data and function icons using Cipetool-provided icon control buttons. There are four control buttons, **constants, arrays, list_images**, and **list_functions**. The constant or array button creates a constant or array icon respectively, and the user must supply the data value(s). When a user wants to access an image file, he/she opens the image icon control menu by pressing the **list_images** button which displays a list of existing images. Upon selecting an image file, an image icon representing the selected image is created. A user may manipulate the location of the icon arbitrarily using the Sun mouse. Similarly, a user can choose a data processing function from the function control window.

The iconic expression illustrated in Figure 2.7 describes the iconic manipulation procedures involved in performing a 3 by 3 box filtering on the image file "girl.bw". First, a user examines the list of existing image files and selects the "girl.bw" using the **list_images** menu. Cipetool creates an image icon representing the girl.bw file. Second, the user selects the filter function using the **list_functions** menu. The filter function icon is created with prompt messages for input image, filter window size, and output storage. Third, the user creates a constant icon for the window size value, 3. Fourth, the user draws lines between the input data icons and the connection boxes to supply the inputs for the filter function. Finally, he/she activates the function. When the filter function is completed, Cipetool displays the output image icon representing the filtered image.

One of the important usages of Cipetool user interface mode is the peripheral device interaction. For example, when a user wants to display an image file in the display device, he/she creates an image icon representing the image file and simply places the image icon in a desired location of the display device icon as shown in Figure 2.7. The image will be displayed in the associated display device on the corresponding location. Such interaction is much simpler than specifying the display location in pixels. The display icon is created when it is allocated. Similar interaction can be applied to other peripheral devices.

2.3.2. Interface to CIPE

Cipetool interacts with CIPE via an interprocess communication mechanism of UNIX. Cipetool converts the iconic user input into CIPE command lines and pipes them to CIPE Cli mode for execution. Cipetool utilizes the function dictionary file for prompt messages of input and output arguments, data types, default values, and help messages for the activated function. The description of the boxfilter function (see Figure 2.7) in the function dictionary is shown below.

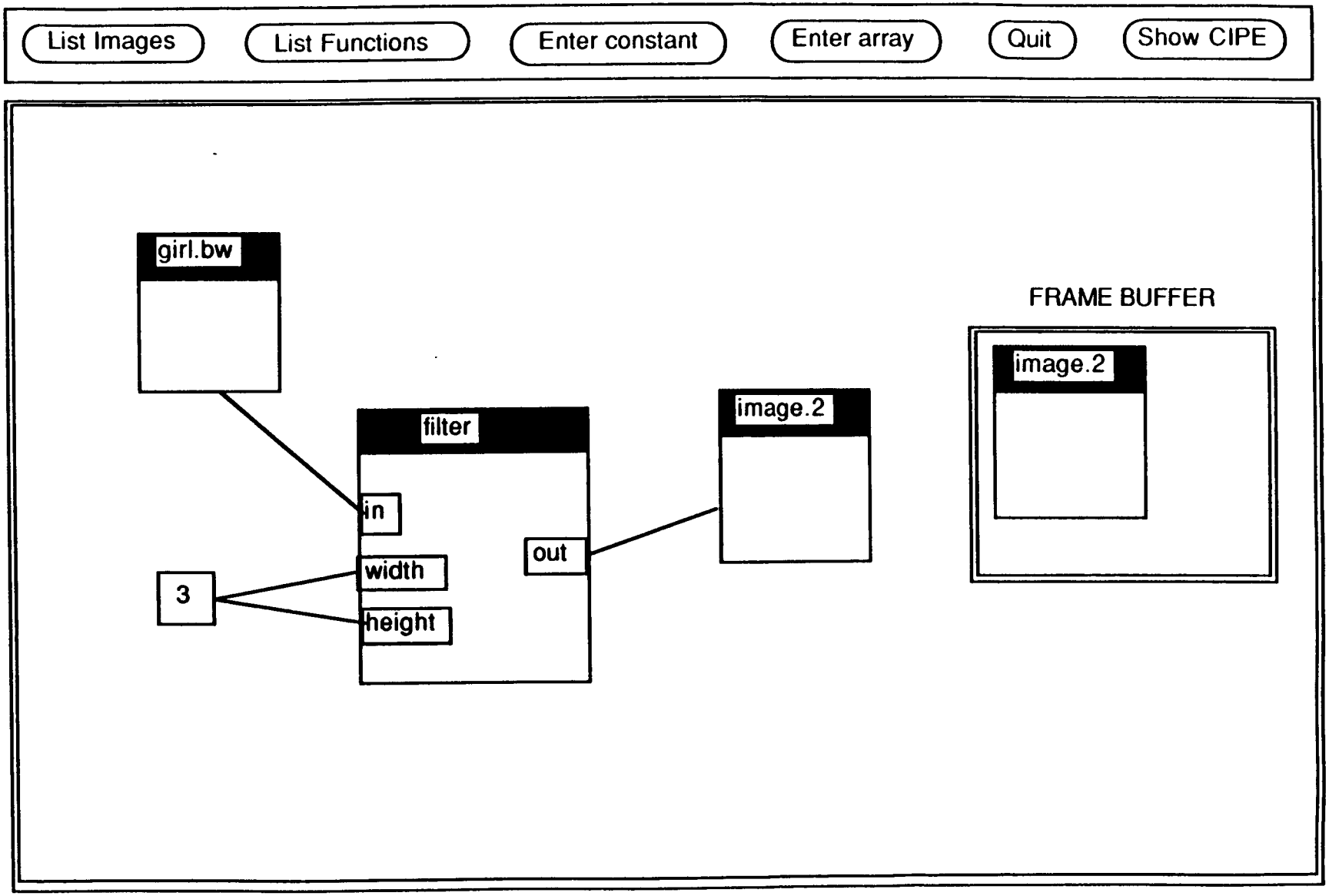

Figure 2.7 Iconic User Interface Example

```
function boxfilter(
    input, "IN", STRING
    input, "Width", INT
    input, "Height", INT
    output, "OUT", STRING)
    help "replaces the pixel with the average intensity of the box"
    pathname "/ufs/cipe/appl/cp/cpbox" uses hypercube
```

In order to incorporate the data representation of CIPE, the data icons must be associated with symbol names. Cipetool employs a symbol name convention by attaching an index to the icon type (constant, array, image). For example, the symbol name of the first image icon is "image1". When the data type is not known, a temporary name is given to the icon. After the filter function is completed, Cipetool checks the description of the output symbol temp from the symbol table. The symbol table content is received from CIPE by issuing the CIPE command **symbols**. Then Cipetool renames the temporary symbol name and displays the output icon using its data type and size. The output of the filter function is renamed to "image2" since the data type is image and it is the second image symbol created in Cipetool. The CIPE Cli command sequence for the iconic expression shown in Figure 2.7 involves the following steps.

1. read (image1) from file (girl.bw)
2. const1 = 3
3. temp = filter(image1,const1,const1)
4. symbols

image1 (girl.bw)	*IMAGE*	*512 lines 512 samples*
temp	*IMAGE*	*512 lines 512 samples*

5. image2 = temp
6. display image2 at 100, 100

2.3.3. Future Direction

Current Cipetool is a prototype version implemented to demonstrate the concept of the graphical user interface The iconic manipulation was found to be somewhat cumbersome to compose a complex command line. The interaction mechanism between CIPE and Cipetool needs to be carefully reviewed to improve the graphical user interface. In order to overcome the limitations of the iconic manipulation, the future Cipetool will implement a complete set of graphical expressions including conditionals, branches, loops, and various procedural descriptors. With the graphical expressions, a user can describe his/her task in a flowchart-like fashion. The graphical expression will be extended to manipulation of a complex system environment with several concurrent systems and various peripheral devices. Most importantly, the future Cipetool will be evolved to be an executive independent front-end process by implementing an

interface layer which will convert the graphical expressions to a proper sequence of command lines for various executives. The virtual executive front-end is an extremely important issue to many existing executives, since it does not require a program conversion process, which is often too expensive.

3. HOST SYSTEM MONITOR

The CIPE system configuration includes a host system and one or more concurrent systems. Thus, the executive is divided into a host resident part and the concurrent system resident part. The host resident part, referred to as a host system monitor, handles overall system interactions and provides generic image processing executive functions. The system management involves peripheral device handling, the concurrent system interface, and application program execution. The current implementation of the host monitor resides in a MASSCOMP 5600 system with an IVAS display device, with near-term extensions planned for Sun-3/Sun-4 workstations.

In order to develop an image processing executive for a virtual concurrent system environment, system setup procedures, data management schemes among multiple systems, and concurrent system interface methods were designed and implemented. A data representation method was also devised to optimize the large volume data manipulation among file system, host system, and concurrent systems. In this section, the host system monitor is described with respect to its four functions: system setup, data management, file management, and concurrent system interface.

3.1. SYSTEM SETUP

The system **setup** function is designed for interactive system configuration. The initial system configuration may be modified using a user-provided *.ciperc* file. The setup parameters include selection of a concurrent system (**none/cube/gapp**) and a display device (**none/ivas/raster/sun**), setting of debug level, session log option, and trace level for program execution. The setup parameters can be entered in a menu mode using a setup entry as shown in Figure 3.1, or at the Cli prompt using individual commands, as shown in Figure 3.2.

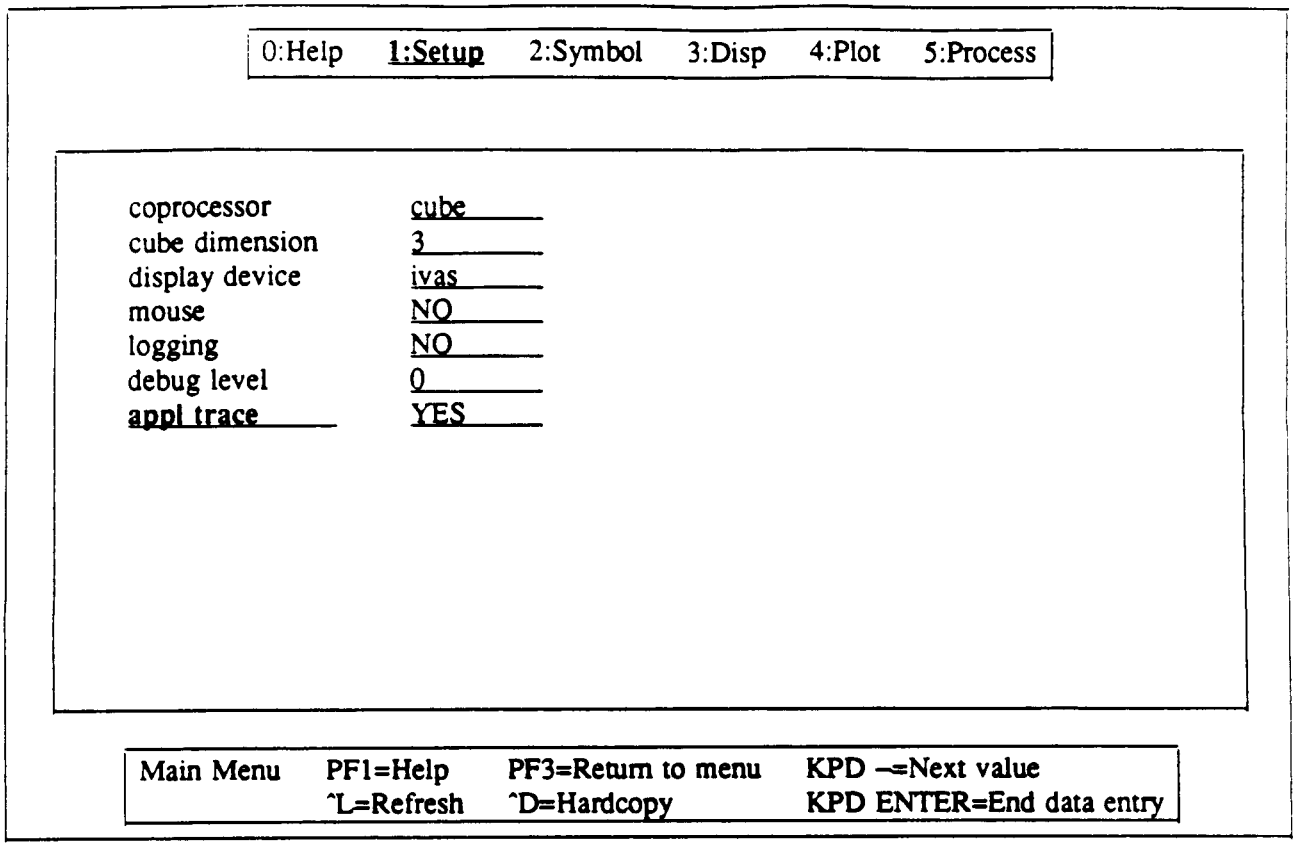

Figure 3.1 Setup Menu

> set coprocessor to cube
> set cube dimension to 3
> set display device to ivas
> turn on appl trace

Figure 3.2 Cli Setup Sequence

3.1.1. .ciperc file

A user provides his/her own default system environment via a .ciperc file in his/her home directory, which is very similar to the .login file or .cshrc file of UNIX. When CIPE is activated, the .ciperc file is read in as a script file and the commands in the file are executed. The example .ciperc file which follows will print "Welcome to CIPE" as CIPE is activated, the main menu will be displayed, and the session will be logged until the session log option is

explicitly turned off.

```
print "Welcome to CIPE"        # print a greeting as CIPE starts
turn on logging                # turn the session log option on
menu                           # set user interface to menu mode
```

3.1.2. Coprocessor Initialization

When a coprocessor is selected, CIPE deallocates any currently allocated coprocessor and allocates the selected coprocessor. The deallocation process warns a user if there are any unsaved data sets before actual deallocation. The Mark IIIfp hypercube, like some other possible coprocessors, is a single user machine. Interactive coprocessor allocation/deallocation allows time sharing of a coprocessor among multiple users.

In the case of hypercube allocation, CIPE invokes the hypercube's Crystalline Operating System (CrOS)* routine, *cubeld(dim,ELT_MONITOR)*, where the variable *dim* has the dimensionality of a hypercube and the *ELT_MONITOR* is the name of the hypercube executive file name. This command resets the hypercube according to the specified dimension and loads the executive in all nodes of the hypercube.

The allocation process for an NCR Geometric/Arithmetic Parallel Processor (GAPP) system consists of loading the GAPP executive on the program memory in the GAPP controller board and the GAPP array memory initialization. The GAPP system executive will be discussed in a separate report in the future.

3.1.3. Display Device Selection/Allocation

A display device can be arbitrarily allocated and deallocated but only one display device can be selected at a given time. The initialization of a display device includes configuration of a video monitor, look-up table composition, a look-up table connection to each image frame, and cursor activation.

The display utilities were developed using the virtual display device interface (VRDI) software developed by MIPL/JPL. The VRDI offers interfaces to most commercially available frame buffers including Ramtek, Raster-tech, Ivas, and Deanza. The CIPE display utilities are described in detail in Section Five.

---

* CrOS is a product of the JPL/Caltech Concurrent Computation Project

3.1.4. Session Logging

The session log provides a record of a user session by copying the standard input and output statements to a session log file. The session log option plays an important role in an interactive image processing environment by keeping track of what a user has done.

A session log file is generated per CIPE session and its name convention is session.log.**pid** where the **pid** is the process identification number of the session. The session log option can be turned on and off during the session freely. When the log option is turned off the session log file is closed with a time tag, and when it is turned back on after a while, the session log file is opened with an update option and the time the log is turned on is recorded.

3.1.5. Trace Flags

A set of trace parameters which can be set for tracing specific execution of functions is installed for optional display of program flow. The installed trace parameters include command line interpreter function-related traces (**parse_trace, lex_trace, codegen_trace, functab_trace**), a symbol table manipulation-related trace (**symtab_trace**), hypercube executive traces (**cube_command, cube_data, cube_symbol**), a general CIPE execution trace (**exec_trace**), and an application program execution trace (**appl_trace**). Only the **appl_trace** can be turned on by both Cli and menu mode for a typical user. All of the other traces can be turned on only in Cli mode and are intended for CIPE system program developers. A debug level is also provided to control the detail of the trace statements.

3.1.6. Function Dictionary

CIPE employs a function dictionary file to support Cipetool and Cli user interface modes. When an application program is installed in CIPE, the program function name and its description are recorded in the function dictionary file by the CIPE system manager. Cipetool displays an icon of a user-requested function with input and output parameter fields, and the Cli parses and verifies the user input command line based on the information provided in the function dictionary.

The function dictionary file contains descriptions of available functions and procedures. The application programs that return output data are defined to be functions and the application programs with no returned value are defined to be procedures. The function description consists of function name, its input argument list, return argument list, and pathname list. The argument structure consists of input or output specification, data type, prompt message, and default value. The pathname shows a corresponding application program name and the applied system where the function can be executed. The procedure description consists of the same items except that there is no return argument list.

FUNCTION *name* (*argument-list*)
RETURNS (*argument-list*)
HELP "*help message*"
PATHNAME *location-list*

PROCEDURE *name* (*argument-list*)
PATHNAME *location-list*

where an *argument-list* is structured as

[*access-type*] [*value-type*] *prompt-message*

for

access-type **INPUT** *or* **OUTPUT**
value-type **INTEGER, FLOAT, DOUBLE, STRING,** *or* **BOOLEAN**
prompt-message "*string*"

and *location-list* is structured as

pathname **USES** *system*

The current implementation is limited to one return argument and one pathname. As various concurrent systems are added to the CIPE environment, each function will have multiple pathnames showing application program file names for corresponding system architectures. Architecture-independent programming can be supported via a function dictionary file where an application program invokes a function, and its architecture-dependent program is activated for a system allocated at the time.

An entry of the CIPE function dictionary is shown in example where the function name is filter and it expects five input arguments including input symbol, window sizes, weight values, and a data distribution type. The function returns an output image. There are two pathnames: one for the filter program that uses hypercube for data processing, and the other filter program that runs in the host system alone. The quoted strings are the prompt messages. A help message describing the function can be provided as well in the function dictionary.

```
function filter(
   input "In",STRING
   input "Box width",INT
   input "Box height",INT
   input "Wgts",INT
   input "Loadtype",STRING)
   returns "Out" STRING
   help "Does a box filter with the given weights"
   pathname "/ufs/cipe/appl/cp/cpfilter" uses hypercube
         "/ufs/cipe/appl/cp/filter" uses host
```

An example procedure description is given below. This particular procedure, which clears a display screen, does not require any input arguments.

```
procedure cleardisplay()
   pathname "/ufs/cipe/appl/cp/cleardisplay" uses host
```

When a function is activated by a user in the iconic interface mode, Cipetool figures out a proper icon size based on the number of parameters, and displays the function icon with prompt messages for the parameters. In the Cli mode, a command line is parsed and verified for the number of arguments and their data types. The function dictionary file will be improved so that an application programmer may not have to deal with user interface modes.

3.2. DATA MANAGEMENT

In an image processing environment, frequently, a sequence of functions is applied to a data set to remove systematic noises and to enhance the data. It is common practice in image processing to apply a function to a data set, save the result to a file, retrieve the file for the next function, and so on until the data set has been completely processed. Such a scenario is extremely time consuming when there is more than one system involved for data processing and each system requires downloading and uploading of data for each function.

In order to achieve efficient data processing with minimized file I/O and host-concurrent system data traffic, a data sharing mechanism among application functions is devised via a cross-referable data management scheme among host monitor, concurrent system monitor, and application functions. In a conventional image processing system, a data set is stored in a file and a file name is used as a reference to a data set. However, in an interactive image processing environment with one or more concurrent systems, a file system is not an adequate way of referring to data sets since a data set may reside in more than one system or may be distributed among several systems. Therefore, CIPE employs a symbol-oriented data management system, where a data set is represented by

a symbol structure.

3.2.1. Symbol structure

The symbol structure is designed so that a data set can be a subset of an existing file, of another symbol, or a set of assigned values. A three-dimensional data set is assumed to incorporate scalar, vector, image, and multi-spectral data. Also, all data types are allowed and data can be stored in a file, in the host memory, and/or in the concurrent system's memory. The current symbol structure assumes that a concurrent system has multiple nodes and local memory like a hypercube. The symbol structure contains the following four types of information.

Data Location - Data location is indicated by a file name if a data set is in the file, a loadmap pointer if it is in the concurrent system, and/or a memory location if it is in the host system memory. When a data set is in more than one location, the data in the most proper location will be accessed depending on the requested function.

Size field - The data size is described in three-dimensional terms: number of lines, samples, and bands. When the data is a subset of an existing file, starting locations are included for each dimension.

Data type - Data types are described as **char, short, int, float,** and **double**. The combination of the data size and the data type allows the proper memory allocation for each symbol.

Data Distribution - When data are distributed among nodes in a concurrent system, a map describing the distribution is composed and attached to the symbol structure. The loadmap is discussed in more detail in Section 5.1.2.

3.2.2. Symbol Assignment

A symbol can be assigned a whole set or a subset of an existing file, the output of a function, a subset of an existing symbol, or a set of immediate values. When a user assigns a value to an existing symbol, CIPE warns the user and the symbol is overwritten if the user ignores the warning.

When a symbol is to be assigned a file or subset of a file, a file name and area parameters are specified for a symbol. The symbol name is checked for redundancy, when the name has not been used before, the symbol is created and its information is copied from the header file. The file name and its relative area that the symbol represents are associated in the symbol structure. The data are not read into a requested system, host memory, or a concurrent system, until a process is requested for the symbol, thus minimizing unnecessary data storage allocation.

```
read A from file_name                          # A is a whole file
read B from file_name(sb,nb,sl,nl,ss,ns)  # B is a subset of a file
```

When a symbol is assigned to the result of a function applied to an existing symbol, the symbol structure is filled in by the applied function based on the the input symbol information. For example, the output of a contrast stretch function will have data size equal to the input while the output of a zoom function will have a zoomed down/up size of input. The memory area for the output result may be allocated in the host system memory or in the concurrent system memory. The details of output symbol memory allocation for the hypercube application program is described in Section 5.1.2.

A user can assign a constant, a vector, or an array of values to a symbol using proper syntax as shown in the example below.

```
A=4
A={1,2,3,4}
A={{1,2,3},{2,3,4},{3,4,5}}
```

The parsed values are stored in the host system memory and their location is stored in the data location field of a symbol. When a symbol is created and the data to be stored in it is a subset of the data in an existing symbol, the data parameters may be expressed in relation to the existing symbol. If a symbol represented by "A" is to be read from a file with a starting line of 100, a starting sample of 100, 256 total lines, and 256 total samples, it may be shown by:

```
read A from "girl.bw"(100,100,256,256)
```

A symbol "B" may be said to be the subset of the data in "A" with a starting line of 100, a starting sample of 100, 100 total lines, and 100 total samples, all by:

```
B = A(100,100,100,100)
```

The data in B has been expressed in relation to the data in A but B's symbol structure will store the data parameters as if B had been assigned data by:

```
read B from "girl.bw"(200,200,100,100)
```

3.2.3. Symbol Table Manipulation

The symbols are managed through a symbol table. The symbol table is a one-dimensional array which contains pointers to the symbol structures. When a symbol is created, its structure pointer is added to the symbol table; when a symbol is deleted, its structure pointer is deleted from the symbol table and the

table is updated. The host monitor provides three symbol manipulation functions for users and application programs to access the symbol table.

create_symbol – When a user assigns (a) value(s) to a symbol and/or reads a file's content to a symbol, the symbol is created by CIPE internally. In Cli mode, the output symbol of an activated function is created by CIPE automatically. CIPE also allows an application program to create a symbol. In general, an application program receives symbol names for input and output where input symbol names refer to existing data sets, and output symbol names refer to data sets that will be produced by the application program. In case of menu mode, the application program receives the symbol name from a user for each output data set and it must create the symbol by calling the create_symbol function with the user-defined symbol name. The create_symbol function creates the symbol entry in the symbol table, allocates a symbol structure, and returns the pointer to it. Each field in the symbol structure must be initialized by the application program for proper size and data type.

get_symbol – Information of a symbol can be retrieved using a symbol name. The symbol table is searched for a given symbol name and the corresponding pointer to the symbol structure is returned. Using the pointer and macro definition of each field of the symbol structure (defined in symbol.h), an application program can access the relevant symbol information. For example, in order to access the number of lines of a symbol A, program statements,

```
a=cipe_get_symbol(A);
ns=CIPENS(a);
```

are required where a is declared as a pointer to a symbol structure.

delete_symbol – A symbol in CIPE can only be deleted by a user, and should be done when a user determines that the symbol is no longer in use. When CIPE receives the symbol deletion command, it checks the data distribution status of the symbol and deletes the symbol from all coprocessors who have any portion of the symbol prior to deleting the symbol from the host monitor. For example, when a user wants to delete a symbol A, the host monitor checks the load type of the symbol A and if the load type is not **none** (the data is distributed in the hypercube), the host monitor sends a **delete_symbol** command to the hypercube monitor prior to deleting the symbol from the host monitor.

3.3. FILE MANAGEMENT

The main purpose of file management is to isolate a programming environment from the operating system and executive-specific file formats so that application programs are unaffected by changes in the operating system and/or file formats.

The disadvantages of executive file management are the added layers of indirection, non-standard (executive dependent) programming, and the requirement of learning an extra set of file I/O routine formats. The CIPE file management tries to avoid these problems by maintaining standard C language I/O function call formats as much as possible and minimizing the indirection. More general purpose data file formats are also devised to be compatible with other executive environments.

3.3.1. Header File Structure

Most executives employ their own file structures using peculiar label formats which require a label conversion/removal process for other programs to access the data. For very large data files (50 Mbytes or more), the label conversion process becomes not only time consuming, but also unfeasible due to limited disk space. Several file structures and header file contents have been evaluated and discussed.

Three major file management schemes were proposed: database utilization for all data files, a separate header file for each data file, and a structured labeling within each file. Since CIPE is designed to demonstrate the computational power of concurrent systems in a generic image processing environment, the operation-dependent labeling and database ideas were rejected. In order to share data sets produced by various executives without the label conversion process, CIPE employs a header file separated from the data file. This section describes the header file design and file I/O routines.

The header file is designed to be a variable length file which can be edited and displayed using standard edit and type operating system commands, *e.g.* vi and cat. The header consists of systematic information, data-set-specific information, and application-dependent information. The systematic information includes an executive indicator, the offset where the data starts in the file, the data size, and the data type. The data-specific information includes starting and ending wavelengths for multi-spectral data. Other information that may be required for a particular application can be stored in the header file. However, the generalized CIPE header processor will not process application-dependent information.

Executive Indicator - The executive indicator field contains the name of the executive for which the file was produced. For example, a file generated by CIPE will have "CIPE" as an executive indicator while a file generated by the VICAR (MIPL/JPL) will have "VICAR" instead. The indicator field is implemented so that the origin of a file can be preserved as well as so that a proper label processor can be activated if the label information should be extracted.

Offset - The offset field contains an integer which indicates where the actual data starts in the data file. This field is implemented so that CIPE can handle data sets with various types of label information. When CIPE

opens a file, it skips the offset number of bytes so that the file index points to the beginning of the data area. The offset field is always zero for the files that are generated by CIPE.

Data size - The data set size is described by the number of bands, number of lines, and the number of samples in a line in order to incorporate three-dimensional data. A scalar value is described as a three-dimensional data set with one band, one line, and one sample. Similarly, a two-dimensional image is described as three-dimensional data with one band. Three-dimensional data sets are stored in a band interleaved fashion where a line from each band is concatenated to form a record in a file.

Datatype - The type of a data set is expressed as **byte, short, int, float,** or **double.** The type information is used for determining data size, for checking arithmetic operation compatibility, and for data conversion.

Others - Besides the systematic and general description of a data set, there can be data-set-related information which may or may not apply for a given application. Such information is an optional part of the header. Currently, the starting and ending wavelength fields are implemented in the CIPE file header for multispectral data sets.

The following example header file shows that this is a CIPE header for a data file "aisa" which contains a 256 byte label, for a three-dimensional multispectral data set of size 200 by 28 by 32. The data type is byte type which ranges between 0 and 255.

aisa.hdr

CIPE
offset = 256
number of lines = 200
number of samples = 28
number of bands = 32
type = byte

3.3.2. CIPE File I/O

The objective of CIPE's file I/O functions is to isolate the programming environment from file structure specifics. CIPE file manipulation consists of an external level with which the programmer interfaces, and an internal level that CIPE uses to resolve file structure-dependent procedures.

For external file I/O CIPE provides a file read function (**cipe_read_from_file**) which reads header file information into a symbol structure and reads the data into the system memory. A user provides a symbol name and file name along with the area of interest. This function activates the CIPE internal file open

function to open the header file and data file, and the read function to read the data.

CIPE also provides a file write function (**cipe_write_to_file**) which writes data associated with a given symbol into a file. This function gathers the data from the hypercube when the data is distributed among the hypercube nodes and composes a header structure. It activates the CIPE internal file create function to create a header file and data file, and the CIPE internal write function to write the data to the data file.

The CIPE internal file I/O consists of file open/create, header file read/write, and data file read/write functions. The file create function (**cipe_create**) creates a data file using a specified file name and also creates a header file. The header file name is composed by concatenating ".hdr" to the data file name. The file open function (**cipe_open**) opens a specified data file and the corresponding header file. This function seeks the actual data starting position in the data file using the offset information of the header file.

CIPE provides a header file read (**cipe_get_image_header**) function which parses the header file information and constructs an image header structure, and a header file write (**cipe_put_image_header**) function which writes the header structure content out to the header file. The read/write/close functions call operating system-provided functions.

3.4. CONCURRENT SYSTEM INTERFACE

CIPE is designed so that it can be applied as a testbed for various types of concurrent systems. The concurrent system testbed requires a unique programming environment which allows a programmer to access function level routines that are shielded from architectural complexities while each function is being executed by a specific concurrent architecture using its full potential. Such a programming environment was approached by implementing a concurrent system interface mechanism between a host system and a concurrent system.

Currently, an 8-node JPL/Caltech Mark III hypercube system and 48 by 48 NCR Geometric/Arithmetic Parallel Processor (GAPP) system are implemented in CIPE. A concurrent system is activated by a user command (**set coprocessor to GAPP/CUBE**) or a menu selection. The activated concurrent system dynamically alters the CIPE execution path by loading the corresponding set of modules. A concurrent system may be activated at a given time and may be released and reaccessed within a CIPE session.

The hypercube-related executive software is divided into two parts: the host resident part, and the hypercube resident part. The host resident part of the software interfaces with the rest of the executive via standard subroutine calls and interacts with the hypercube resident part via a set of predefined commands. The hypercube resident part is called the hypercube monitor which waits for a command from the host process as a slave process. The host

resident part of the hypercube software is discussed here with respect to command protocol, data distribution, and application program execution.

3.4.1. Hypercube Command Protocol

The host system monitor and the hypercube monitor have a master and slave relationship. The host monitor issues a command and the hypercube monitor performs the command. The common program environment for generic image processing applications is investigated to design a set of relevant commands and their execution protocols. Within CIPE, a hypercube application program is viewed as a routine that performs a specific data processing function on existing data sets. A typical scenario that takes place in the host system monitor when a hypercube application program is activated is illustrated below.

Prepare input and output data sets
(1) create the input symbols in hypercube,
(2) compose and download the load maps for the input symbols,
(3) download the data according to the load maps,
(4) create the output symbols in the host,
(5) create the output symbols in the hypercube,
(6) compose and download the load maps for the output symbols.

Execute the function
(1) download the function module,
(2) execute the function,
(3) wait till execution is completed.

In order to perform interactive hypercube program execution with automatic data preparation, CIPE uses eleven commands to communicate with the hypercube monitor with respect to symbol management, data I/O, and program execution. The commands are sent from the host to the hypercube with an optional acknowledgment request. When the request is sent, the hypercube sends back an acknowledgment of whether the request was granted or not. Once the command is granted, required information for the individual command is passed to the hypercube monitor according to the predefined sequence.

For symbol management, the host executive and the hypercube manage data using a symbol table, as discussed in Section 3.2. The hypercube executive receives a command for symbol creation (CREATE_SYMBOL) and deletion from the host executive (DELETE_SYMBOL) for the symbols whose data are provided and/or manipulated by the host.

The data are distributed to the hypercube nodes after a specific load map is written to the hypercube (READ_LOADMAP, READ_DATA). The load map describes the data area that each node will receive and is determined by the application function. When the data are read back from the hypercube, the load map directs the way the data should be assembled (WRITE_LOADMAP,

WRITE_DATA). When the data are already present in the hypercube, but need to be distributed differently, a data redistribution request (REDIST_DATA) may be issued.

To manage programs CIPE employs a concept called incremental program loading, where an application function is loaded for execution dynamically. To support the dynamic program execution, CIPE sends a command to load an executable module (LOAD_MODULE) and a command to execute the module (EXECUTE_MODULE). A set of built-in functions are also provided and are resident in the hypercube during a CIPE session. The built-in functions can be activated using the EXECUTE_BUILTIN command.

When a user wishes to terminate a CIPE session or to change the coprocessor from hypercube to another processor, CIPE sends the EXIT command to the hypercube monitor. The hypercube may be reaccessed by the coprocessor selection command.

3.4.2. Application Program Management

As mentioned above, an application program for a hypercube coprocessor consists of a host resident part and a hypercube resident part. When an application function is activated in CIPE, the host system monitor loads the host resident application program module into the preassigned program area and executes the program. The host resident program loads the corresponding hypercube program to the hypercube (incremental loading by the hypercube resident monitor) after the data to be processed are properly distributed in the hypercube. The incremental loading of an application program is made possible by linking the program with CIPE to resolve all of the global references. More details on incremental loading are available in Section 4.3, and a Unix manual on cc.

3.4.3. Hypercube Data Distribution

The image processing environment involves large volumes of data. Utilization of a hypercube system with many nodes significantly enhances the computation by processing the data concurrently. However, large data volume manipulation is somewhat complicated since the local memory architecture does not allow direct data sharing among nodes, and the data must be transferred via interconnecting channels.

In order to minimize the application programmer's effort in data manipulation, and to allow a very flexible data distribution scheme, a load map structure is composed where a programmer can specify an exact area in a given data set for each node. A three-dimensional data set is assumed and a subarea is described by starting and ending points in each dimension (line, sample, and band).

CIPE also provides a set of standard load maps for the frequently applied data distribution schemes. The currently supported standard distribution types are a broadcast distribution (BCAST_DIST), a horizontal distribution

(HORIZ_DIST), a vertical distribution (VERT_DIST), and a grid distribution (GRID_DIST). An application programmer simply requests a proper distribution type, and the load map is automatically composed.

The data can be redistributed from one standard distribution to another standard distribution without involving the host data download, except to BCAST_DIST. Data distributions other than the standard distribution types are considered to be customized data distributions (CUSTOM_DIST) and require a user-composed load map for each node. The customized data distribution cannot take advantage of hypercube internal data redistribution. The actual data distribution and redistribution are discussed in more detail in Section 4.2.3.

3.4.4. Data Distribution in an Application Program

An application program determines how the data should be distributed and calls a data distribution routine (**cipe_write_data**) with a symbol name and a distribution type. When the requested distribution type is one of the standard types, CIPE calls a load map composition routine (**cipe_compose_loadmap**). Otherwise, it assumes the load map is composed correctly by the application programmer. Then, CIPE checks to see if the data has been downloaded to the hypercube previously. If the data have not been downloaded, the data will be downloaded according to the load map.

If the data have been downloaded previously, there are three possible distribution cases: (1) the data distribution is same as requested - no need to download again; (2) the new distribution type is compatible with the old type (i.e. one of standard types) - call redistribution routine (**cipe_redist_data**) with old load map and new load map information; and (3) the new distribution type is not compatible with the old type - delete the existing symbol from the hypercube and recreate the symbol using the new data distribution load map. In the last case the data exists only in the hypercube; the data will be read back to the host prior to deletion.

4. HYPERCUBE RESIDENT MONITOR

A JPL/Caltech MARKIIIfp hypercube is employed as a concurrent coprocessor for CIPE. Each node is equipped with 4 Mbyte of memory, a Weitek floating point processor board, a Motorola 68020 I/O processor, and a Motorola 68020/68881 data processor. The implementation details of CIPE with regard to the hypercube are specific to the MARK III/CrOS hypercube, but the basic concept associated with utilization of a multi-node interconnection topology, localized memory system, and multiple programming capacity are shared among all MIMD systems. Therefore, the monitor design can be easily generalized to other MIMD systems.

The localized memory configuration of a hypercube system allows a very high level of concurrency, since a large number of nodes can be connected without creating a memory access bottleneck. However, three problems arise: distribution of data among multiple nodes is significantly more complicated; data transfer between a host system and a hypercube system performed through a single hypercube node, node 0, creates a bottleneck; and the memory is shared by program and data, one affecting the other for the limited memory space. All of these are serious limitations for image processing applications where large data sets are commonplace.

To aid in distributing data to the hypercube, CIPE supports a set of standard data distributions and an easy-to-compose data load map structure for customized data distributions. These significantly simplify the programming effort required in hypercube data manipulation.

CIPE employs a global data management scheme to permit data sharing among successive application programs which reduces the data transfer between the host and the hypercube. Data can be redistributed from some distribution types to others within the hypercube using the hypercube interconnection topology. Redistributing data within the cube reduces data traffic between the host and the hypercube, significantly, by allowing data to be shared among successive application programs with different data distribution requirements. Near-term enhancements to the hypercube include concurrent I/O capabilities between the host and multiple nodes of the hypercube. This enhancement should significantly reduce the data transfer bottleneck.

A specialized program management method, incremental loading, is devised for efficient hypercube memory utilization for program and data. When an entire data set cannot be present in the hypercube at one time, another level of data distribution difficulty is added, and the programming is significantly complicated. Moreover, only the activated function module is required to be present in the hypercube node for data processing. Therefore, a trade-off has been made between the program area and data area. By loading only an activated function module to a preallocated program area, the most memory may be used for data.

The hypercube monitor performs these data and program management functions upon the host system monitor's command. There are eleven commands including seven data management related commands, three program management related commands, and one termination command. In this section, the execution of these commands is examined in detail.

4.1. INTERFACE TO HOST EXECUTIVE

When a CIPE user initializes the hypercube as the coprocessor of choice, the hypercube is reset and the hypercube monitor software is downloaded to the cube. Each node of the hypercube has an identical copy of the monitor and this monitor will remain resident in each node of the hypercube throughout the use of the hypercube as a coprocessor. The host and hypercube interact as a master and slave where the hypercube monitor waits to receive a command from the host, acknowledges its receipt, executes it, and waits for the next command.

The monitor interfaces to the host executive providing for the execution of eleven commands as mentioned in Section 3.4. The commands are for program and data management including **create_symbol** and **delete_symbol**, for symbol management, **read_load_map, read_data, write_load_map, write_data**, and **redist_data** for data distribution, and **load_module, execute_module** and **execute_builtin** for program management. For each received command, the hypercube monitor and the host monitor follow predetermined communication steps to receive/transfer necessary information for the command execution. An execution sequence is described below, using a **create_symbol** command as an example.

Host Monitor	**Hypercube Monitor**
1. Send symbol name	**1. Receive a symbol name.**
2. Wait for acknowledgment	**2. Check if the symbol exists already. If so, then send error acknowledgment (NACK) and go back to wait; otherwise, send acknowledgment** (ACK). **Create the symbol entry in the symbol table.**
3. If NACK abort the command	**3. Receive the data size and allocate the data area; else, send data size.**

Create_symbol Execution Sequence

4.2. DATA MANAGEMENT

A data set is distributed among nodes in the hypercube for processing. Each node may have the entire data set or a subarea of the data set according to the processing need. The data set distribution information is composed by the host monitor for each node and passed down to the hypercube monitor prior to actual data transfer. A data set mentioned in this Section refers to a portion of data which resides in a given node.

Similar to the data management of the host monitor discussed in Section 3.2, a data set in the hypercube monitor is represented as a symbol which is associated with an attribute structure describing its name, data type, and load map (distribution scheme). A data set is accessed by its symbol name by application programs. The hypercube monitor employs a symbol management scheme very similar to the host monitor.

4.2.1. Symbol Structure

The symbol structure employed by the hypercube monitor receives the symbol information through a **create_symbol** command for the name and datatype, and through a **read_load_map** command for the loadtype and data distribution scheme. The symbol structure contains name, datatype, and a load map field. The load map field contains loadtype, location, and size field. The symbol structure declaration can be found in Appendix I under *elt_symbol.h*.

Data type - An element within a symbol is described for its type as **char, short, int, float, and double**. The combination of the data size and the data type allows a proper memory allocation for each symbol.

Load type - Data distribution type is described as **BCAST_DIST, HORIZ_DIST, GRID_DIST, VERT_DIST, CUSTOM_DIST** (see Section 3.4 for the type definition). The loadtype information combined with the load map allows the data redistribution within the hypercube.

Load map - The load map describes the absolute data location of a symbol in a node in reference to the whole data set in three-dimensional terms. The starting location and its size for each dimension are expressed in lines, samples, and bands.

Data - Data area is allocated separately from the symbol attribute structure, and its address is stored in the data location field of the symbol structure.

4.2.2. Symbol Table

The symbols are managed through a symbol table. The symbol table is a one-dimensional array which contains the pointers to symbol structures. When a symbol is created, its structure pointer is added to the symbol table. When a

symbol is deleted, its structure pointer is deleted from the symbol table and the table is updated. The hypercube monitor provides three symbol manipulation functions for hypercube module application programs to access the symbol table.

create_symbol - A symbol can be created as a global symbol or as a local symbol. A global symbol is created when the host monitor sends a **create_symbol** command. The global symbols are employed for data sharing among application programs. The global symbol data is managed by the hypercube monitor and only application programs can access them. A local symbol is created by an application program and is known only to the application program. Local symbols are deleted by the monitor upon the completion of the hypercube resident application program which created them. When a local symbol is created, the hypercube monitor allocates a symbol structure, initializes each field with default values, and returns the pointer to the allocated area. Each field in the symbol structure must be updated by the application program for proper size and data type.

get_symbol - Symbol information is retrieved using a symbol name. The symbol table is searched for a given symbol name and the corresponding pointer to the symbol structure is returned. Using the pointer and macro definition of each field of the symbol structure (defined in *elt_symbol.h*), an application program can access the relevant symbol information. For example, in order to access the number of lines of a global symbol A, program statements

```
a=get_symbol(A,GLOBAL);
ns=NS(a);
```

are required.

delete_symbol - a symbol entry created by an application program may be deleted by an application program using the **delete_symbol** function. Symbols created by the host monitor are deleted by the hypercube monitor when the **delete_symbol** command is received.

4.2.3. Data Distribution/Redistribution

Data is distributed through the use of the monitor's READ_DATA and WRITE_DATA commands. Before data may be read, a symbol must exist and a load map must have been read to know the data decomposition type and the specific load map parameters. Data is read into the cube in one of five distribution types: a BCAST_DIST, which sends the entire data set to each node; a HORIZ_DIST, which breaks the data into horizontal regions; a GRID_DIST, which breaks the data into a grid; a VERT_DIST, which decomposes the data into vertical regions; or a CUSTOM_DIST, which permits the decomposition of data in any manner that may be specified by six parameters, one each for the

starting point in the x,y,and z directions, and one each for the duration of pixels in each direction.

CIPE images are stored line by line. This means that a two-dimensional image with dimensions 512 x 512 pixels is stored, line 1 first, with samples 1 through 512, then line 2, samples 1 through 512, continuing to line 512. The host computer communicates data and images to the hypercube using commands provided by the Crystalline Operating System (CrOS). The hypercube command, **bcastcp** (broadcast from the control processor), allows a user to specify the address of a buffer and the number of bytes to be sent from that buffer. The specified number of bytes is sent from the beginning of the buffer to each node, so that each node has exactly the same data. This is the way a BCAST_DIST is achieved. Another hypercube command, **mloadcp**, allows the user to specify the address of a buffer and a load map of how many bytes are to go from that buffer to each node. The command starts at the beginning of the buffer specified and sends distinct but contiguous blocks of data to the nodes, beginning with node 0. In this manner an image ends up in the nodes in a horizontal decomposition (Figure 4.1), and no two nodes receive the same data.

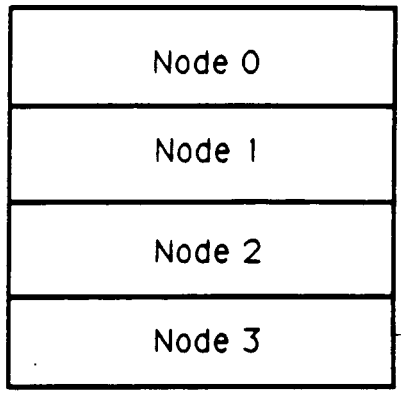

Figure 4.1 Horizontal Decomposition

Neither a GRID_DIST, VERT_DIST, or CUSTOM_DIST are achieved simply by the use of a CrOS communication call. A grid or vertical decomposition begins by being downloaded into a horizontal distribution. The data is then moved within the cube to create a grid or vertical decomposition. The same result could be achieved by rearranging the data in the host and then using the same hypercube command as for a horizontal distribution, mloadcp, to send the data to the cube. Such an implementation does not allow the user to take advantage of the additional processing capability of the hypercube. The bold lines in the following diagram outline the areas of an image held by each node in a horizontal distribution. The patterned regions show the data regions desired to form a grid decomposition (Figure 4.2). To determine how to

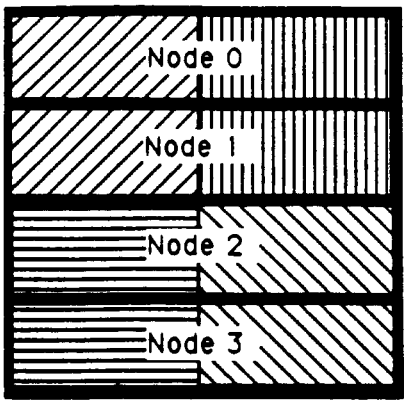

Figure 4.2 Horizontal Decomposition and Desired Grid Decomposition

manipulate the data from one decomposition to another, it is important to observe the node connectivity within the cube, and the possible data paths between nodes for data exchange (Figure 4.3).

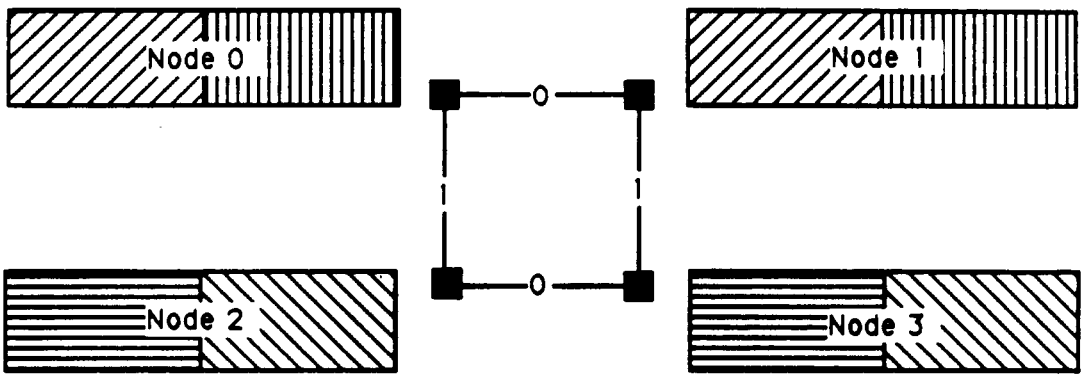

Figure 4.3 Hypercube Node Connectivity

Figure 4.3 shows the connectivity of the nodes and the data each node is holding. In a four node hypercube only nodes 1 and 2, and nodes 0 and 3 lack a direct connection. In this example the data exchange to achieve a grid decomposition is simple. Nodes 0 and 1 in a horizontal decomposition have all the data necessary to create two regions of a grid decomposition. The same is true for nodes 2 and 3. Moreover nodes 0 and 1, and nodes 2 and 3 are directly connected. All that is necessary is for each node to determine what data it needs to keep, what data it needs to exchange, and to then utilize the appropriate CrOS command to exchange the data with the appropriate neighbor.

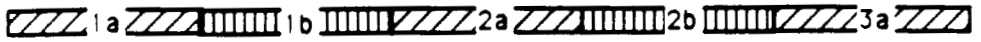

Figure 4.4 Data in Node O

The beginning of Node 0's data as it is stored in memory is pictured in Figure 4.4. The task of separating the data to be retained and the data to be exchanged requires that an additional buffer half the size of the current buffer be created. Each node knows the length of its lines from its load map. Node 0 will retain the first half of its first line, 1a, in the beginning of the original buffer. The second half of its first line, 1b, will be copied to the beginning of the new buffer. The area in memory where 1b was stored in the original buffer will not be erased, but other data will be written on top of it. The first half of the second line, 2a, will be appended to 1a in the original buffer. The second half of the second line, 2b, will be appended to 1b in the new buffer. The contents of the new and original data buffers, at this time, are pictured below (Figure 4.5).

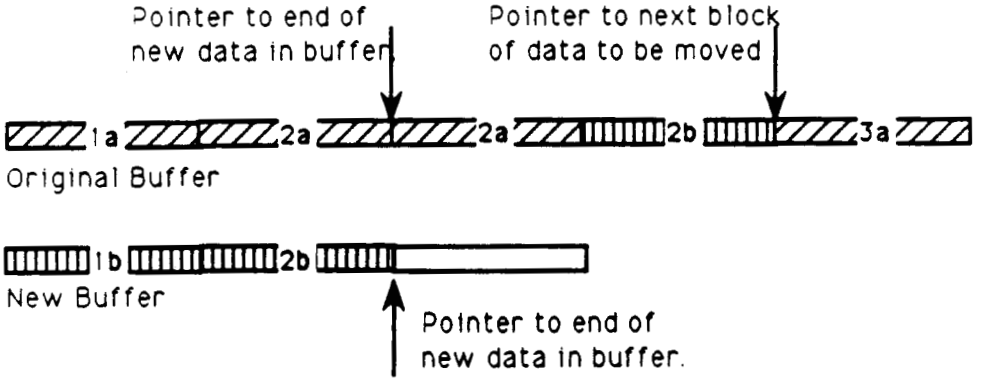

Figure 4.5 Node 0 and Node 1 Buffer Contents

Two pointers are required in the original buffer: one to maintain the end of the contents of the new data being compacted there, and one to maintain what data is now being moved. The process occurs in each node until all the data in the original buffer is divided between the two buffers.

As described earlier, nodes 0 and 1, and nodes 2 and 3, need to exchange data to create a grid decomposition, and these nodes are directly connected and require no middle node to communicate. No CrOS command allows a user to

specify a node with whom to exchange data. A programmer must determine what channel connects two nodes and request that an exchange be done on that specific channel. The channel connecting two nodes is determined by looking at the binary gray code representation of the node numbers and seeing in which one digit the two nodes desiring to exchange differ. Node 0 may be represented in binary by 000, and node 1 by 001. The units, 2's, and 4's digits are numbered from right to left beginning with 0. So 000 and 001 differ in one digit, the units digit, numbered 0, and therefore communicate along channel 0. Similarly node 2, 010, and node 3, 011, differ in the units digit, and also communicate along channel 0. Therefore, in order for a horizontal distribution to become a grid decomposition, all nodes must determine the data they need to retain and exchange, and exchange the appropriate data with their neighbor along channel 0. (Figure 4.3 has channel numbers between nodes labeled.) It may be seen empirically that this process is true for any n-dimension hypercube.

To create a vertical distribution from a horizontal distribution requires that data be exchanged along successive channels, from 0 through the number of dimensions of the hypercube. In each case, the nodes determine data to be exchanged and retained, divide it up, and exchange on the appropriate channel. The process may also be reversed to go from a vertical to a horizontal distribution. Other uses of this process to move forward and backward through a horizontal, grid, or vertical decomposition will be discussed later.

A **CUSTOM_DIST**, or a customized distribution, makes full use of its load map. An entire image is broadcast to each node in chunks using the CrOS routine, bcastcp, discussed earlier. Before each data chunk is sent, a package is sent to each node telling it what part of the image to expect. Each node looks at its own load map to determine what portion of the incoming data it is supposed to have. It copies any of the incoming data it needs to its own buffer, and then prepares to accept the next package of data information and data. The size of the data chunks being sent is now rather arbitrary. In the future, the size of the data chunks sent will be determined by the available memory in the nodes.

In addition to providing for the initial distribution of data, CIPE provides some capabilities for redistributing data from one decomposition to another. These capabilities help to achieve one of the goals of CIPE, to minimize data I/O and thus improve the overall efficiency of the image processing being done. Different image processing applications may prefer data in one decomposition over another. If CIPE is able to redistribute data within the hypercube without the need to download data again it is able to save time. It is not always possible to manipulate data within the hypercube from one decomposition to another. The following diagram illustrates what redistribution cases are possible (Figure 4.6). If there is a path created by the directional arrows between two distribution cases that have no direct arrow between them, as in the **HORIZ_DIST** to **GRID_DIST** to **VERT_DIST**, that redistribution is possible

as well, as in **HORIZ_DIST** to **VERT_DIST**.

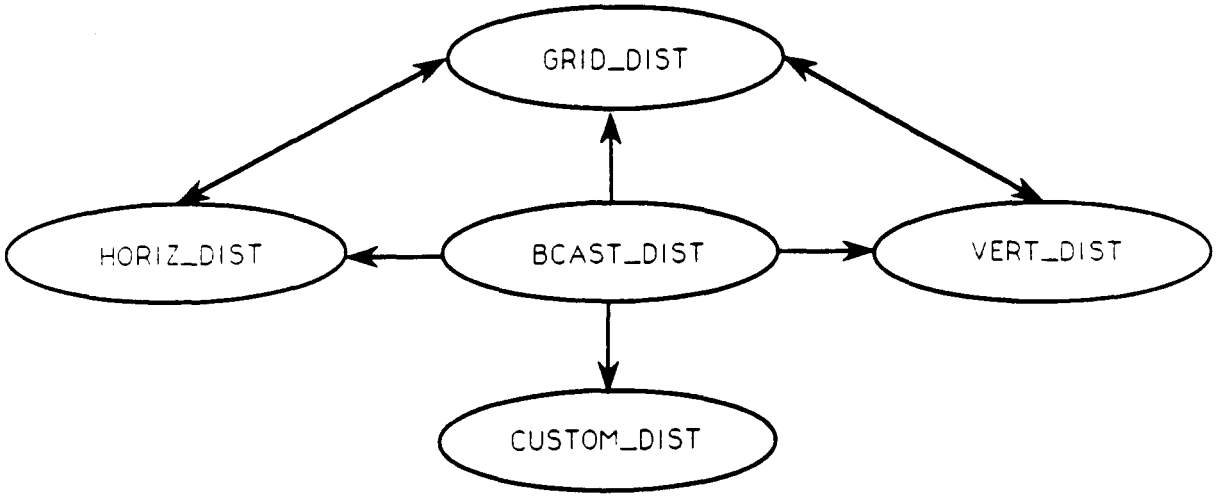

Figure 4.6 Possible Redistribution Cases

Moving HORIZ_DIST to GRID_DIST, and GRID_DIST to VERT_DIST, is achieved exactly as it is for an initial grid or vertical distribution as described before. Moving from a VERT_DIST to a GRID_DIST, and from a GRID_DIST to a HORIZ_DIST, is achieved, similarly, by reversing the process. Data is exchanged along channel n where n is the number of dimensions of the hypercube. A node now has its own buffer and another it received from a neighboring node. Then two buffers are pieced together, one line from each of two buffers, to create one line of a new buffer. This process continues along each channel decreasingly to channel 0. The process may stop at any channel to create some variety of a grid decomposition. Depending on the dimension of the hypercube, more than one kind of a grid distribution may be formed between a horizontal and vertical distribution. What CIPE calls a GRID_DIST is one data exchange along channel 0 beyond a HORIZ_DIST. Redistributing data from a BCAST_DIST to any other distribution is achieved similar to the way a CUSTOM_DIST is achieved. All nodes have the data for an entire image, and they know the dimensions of that image. The nodes receive their new load map indicating what data they should now have, and copy the appropriate data to the beginning of their buffer space, overwriting the previous data.

CIPE's methods for distributing and redistributing data for the hypercube have been developed to use existing CrOS routines to maintain compatibility with others using the Caltech/JPL Mark III hypercube, and to avoid the need to exchange data between two nodes that are not directly connected. If two nodes needing to exchange data have no direct connection, extra memory and processing time is needed in the nodes to maintain their own data as well as permit

data to pass through enroute to another node. Using extra memory for this purpose places greater limitations on our program and/or data size, thus diminishing the effectiveness of CIPE. One problem with the method of distributing and redistributing data CIPE uses is that in every decomposition there is some place in which contiguous pieces of an image are stored in non-adjacent nodes. In the following diagram the middle of the image, held by nodes 1 and 2, is patterned (Figure 4.7).

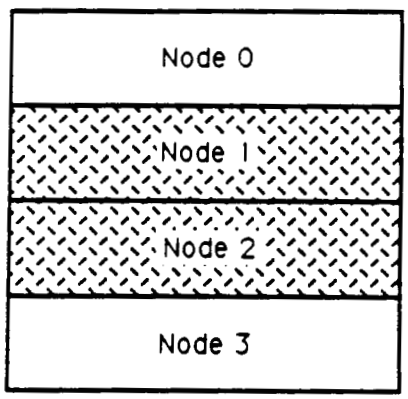

Figure 4.7 Node Ownership of Image Sections

But as the data is held in the nodes the patterned region is in two non-neighboring nodes (Figure 4.8).

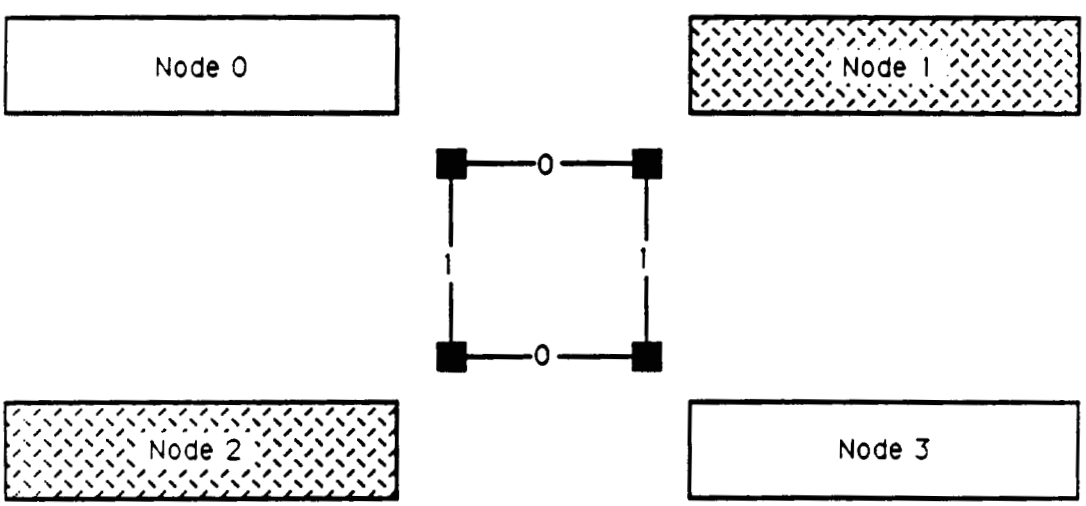

Figure 4.8 Contiguous Data in Non-adjacent Hypercube Nodes

This is a real weakness because many image processing applications will require overlapping data regions. Since no CrOS command provides for down-loading data so that data regions overlap, it would be convenient to be able to download data into one of CIPE's standard distributions and then exchange the necessary data so the data regions overlap. Not only would it require exchanging data with a non-neighboring node and the complications that have been explained which that involves, but it also requires a tricky routing algorithm which has not yet been determined.

4.3. PROGRAM MANAGEMENT

The hypercube monitor provides 100 kbyte of memory area for an application program under the name of Progarea. Hypercube modules for each application program are linked with the hypercube monitor so that each is located precisely in the Progarea using an incremental loading facility of the UNIX linker. The linker resolves the external references of each application module and produces an executable file.

Upon receiving a **load_module** command, the hypercube monitor reads the executable file into the Progarea. When the host monitor issues an **execute_module** command, the hypercube monitor jumps to the Progarea and starts executing the loaded module. Thus, the incremental loading mechanism limits the hypercube resident application module memory usage to 100 kbyte.

5. APPLICATION PROGRAMMING WITH CIPE

The CIPE programming environment pursues an architecture-independent environment where a programmer may write an application program without concern for architectural specifics. This section presents application programming with CIPE using the JPL/Caltech Mark III hypercube as a concurrent system. Without the support of an executive or the use of a concurrent system, an application program must perform three functions: user interface, data I/O, and data processing. In an application program using a hypercube, the data processing is downloaded to the hypercube system for concurrent processing. In this case, a programmer writes two program modules, one for the host and one for the hypercube. In the host module, the programmer is responsible for reading data from a file and distributing it among the hypercube nodes. It is also the programmer's responsibility to receive the data in the hypercube module. This process requires a full knowledge of the hypercube architecture and operating system.

CIPE provides simple function calls to support user interfaces, perform file I/O and/or data I/O, and various other functions. An overview of a CIPE application program is illustrated in Figure 5.1. The host module consists of three parts: the user interface which interacts with the Menu and the Cli modes of CIPE; the interaction with the host system monitor to perform input data distribution, output data load map distribution, and hypercube module activation; and finally, interaction with the hypercube module to pass parameters to the activated hypercube module and wait for completion of the data processing. The hypercube module consists of two parts: the parameter reception and data processing. Data is received prior to hypercube program module activation by interaction between the host system monitor and the hypercube resident monitor. The hypercube module accesses the data by interaction with the hypercube monitor.

The design and implementation details of the user interface, host system monitor functions, and hypercube monitor functions are described in previous sections. This section explains how an application programmer can utilize the CIPE functions to process data in a concurrent system environment.

5.1. HOST SYSTEM RESIDENT MODULE

In CIPE, a host resident application program can be viewed as a subroutine dynamically loaded to a fixed location where CIPE is the main program. When a user requests a function provided by the application program the main program loads an application program file and activates it by a subroutine call mechanism. A set of common program procedures is shared by all of the host system resident modules in CIPE. Each module must provide two user interface modes for user input parameters, input data access and distribution, and hypercube module activation and interaction. As illustrated in the host resident application program example, Figure 5.2, the host resident module of a filter

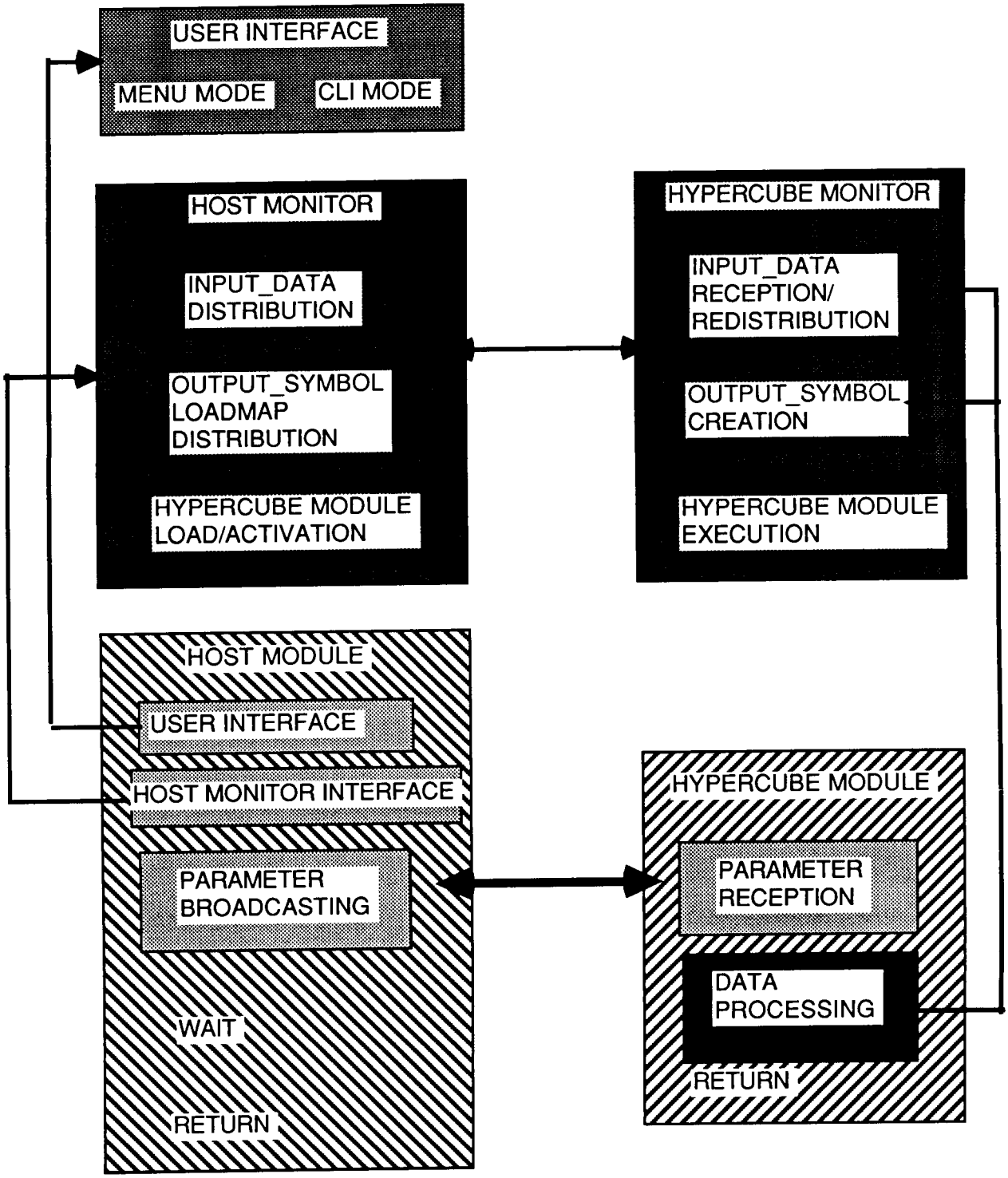

Figure 5.1 Hypercube Programming in CIPE

program consists of four sections: the variable declaration, the user interface, the host resident monitor interaction for data distribution and hypercube module activation, and input parameter passing to the hypercube module. The examples referred to throughout the subsections of 5.1 refer to Figure 5.2. All host resident modules must include *cipeappl.h* to provide the variable and structure definitions common to all CIPE applications.

5.1.1. User Interface

CIPE can be in one of three user interface modes (menu mode, Cli mode, or Cipetool) when an application program is invoked. Since Cipetool translates a user's actions into Cli commands, a CIPE application program must provide two user interface modes for user input parameters using provided function calls and macro definitions. Future enhancements to CIPE will include a consistent means for a programmer to provide all three user interfaces, and will eliminate the need for function calls. The main difference between the menu mode and Cli mode user interfaces is in the user parameter value reception process.

5.1.1.1. Creation of a Menu Interface

In menu mode, parameters are received directly from a user and stored in their corresponding variables by the use of the Yamm routine **def**. Interactive error checking of parameters is the responsibility of the programmer and is accomplished by the use of the Yamm routine **getpar**. A programmer may provide interactive help messages for each parameter field in a manner similar to the error checking. Both **def** and **getpar** must be used to create the desired display on the input screen and to read the user's input; **getpar** must follow **def**. To be able to use the routines **def** and **getpar**, an applications programmer must first make the declarations:

```
#ifndef lint
static char sccsid[]="%W% %G%";
#endif lint

/* %M% version %I%, %G% */
#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <cros.h>
#include "cipeappl.h"

cpfilter()                                                    cpfilter
{

/*************************************************************************/
/*****************  VARIABLE DECLARATION SECTION  ********************/
/*************************************************************************/
/*------------------------------------------------------------ Menu Related */
    struct param params[4];
    int param_count;
    int check_filter();

/*------------------------------------------------------------ Symbol Related */
    cipe_sym_name input,output;
    struct cipe_symbol *sin, *sout, *cipe_get_symbol();
    int sindex;

/*------------------------------------------------------------ Hypercube Related */
    int *packets,packet_size;
    int *bufmap_receive, *bufmap_send;

/*------------------------------------------------------------ Application Related */
    int window[2];
    char loadtype[2];
    int i, ok, share, *weight, *argptr;


/*------------------------------------------------------------ Variable Initialization */
    bufmap_receive = (int *)malloc((nproc+1)*sizeof(int));
    bufmap_send = (int *)malloc((nproc+1)*sizeof(int));
    bufmap_send[0] = nproc;
    bufmap_receive[0] = nproc;

    strcpy(loadtype,"c");


/*************************************************************************/
/*********************  USER INTERFACE SECTION  **************************/
/*************************************************************************/
/*------------------------------------------------------------ Menu */
    if (MENU_MODE) {
        param_count = 4;
        def(params,0,"input symbol name          ",STRING,input,WID,32,LINE,2,
        REQ,END);
        def(params,1,"output symbol name         ",STRING,output,WID,32,LINE,3,
        REQ,END);
        def(params,2,"filter window(nlw,nsw)  ",INT,window,INCR,4,WID,10,
        LINE,4,DUP,2,REQ,END);
        def(params,3,"load type (h/v/g/c)       ",STRING,loadtype,WID,10,
        LINE,6,REQ,END);
        if (getpar(params,param_count,check_filter,NOHELP)) return;

        weight = (int *)malloc(window[0]*window[1]*sizeof(int));
```

Figure 5.2 Spatial Filter Control Processor Program

```
        for (i=0; i<window[1]; i++) {
            def(params,i,"weight values ",INT,weight+i,
            INCR,sizeof(int)*window[1],LINE,2,
            ENTRYLINE,3,ENTRYCOL,i*10,DUP,window[1],GROUP,0,REQ,END);
        }
        if (getpar(params,window[1],NOECHK,NOHELP)) return;
    }
/*------------------------------------------------------------ End_Menu */

/*------------------------------------------------------------ Cli */
    else {
        sin = ARGPTR(0);
        window[0] = *(int *)ARGDATA(1);
        window[1] = *(int *)ARGDATA(2);
        weight = (int *)malloc(window[0]*window[1]*sizeof(int));
        argptr = (int *)ARGDATA(3);
        for (i=0; i<window[0]*window[1]; i++)
            weight[i] = argptr[i];
        strcpy(loadtype,(char *)ARGDATA(4));
        sout = RESULTP;
    }
/*------------------------------------------------------------ End_Cli */

/*****************************************************************************/
/********** CIPE HOST RESIDENT MONITOR INTERFACE SECTION **********/
/*****************************************************************************/
/*------------------------------------------------------------ Symbol Retrieval */
if (MENU_MODE) {
    sin = cipe_get_symbol(input);
    ok = cipe_creat_symbol(output,&sout,&sindex);
}
/*------------------------------------------------------------ End_Symbol Retrieval */

/*------------------------------------------------------------ Custom_Dist */

    share = (CIPENL(sin)+nproc-1) /nproc;

    if (strcmp(loadtype,"c")==0) {
        if (CIPELOADMAP(sin) == NULL)
            CIPELOADMAP(sin) =
            (struct loadmap *)malloc(nproc*sizeof (struct loadmap));
        for (i=0; i<nproc; i++) {
            CIPELOADMAP(sin)[i].ss = 1;
            CIPELOADMAP(sin)[i].sl = i*share - (window[0] /2);
            CIPELOADMAP(sin)[i].sb = 1;
            CIPELOADMAP(sin)[i].nl = share+window[0];
            CIPELOADMAP(sin)[i].ns = CIPENS(sin);
            CIPELOADMAP(sin)[i].nb = CIPENB(sin);
        }
        /* starting line must be positive value */
        CIPELOADMAP(sin)[0].sl = 1;
        /* The last proc may get little less data */
        CIPELOADMAP(sin)[nproc-1].nl =
        CIPENL(sin)-CIPELOADMAP(sin)[nproc-1].sl+1;
        ok = cipe_cube_write_data(sin,CUSTOM_DIST);
    }
/*------------------------------------------------------------ End_Custom_Dist */

/*------------------------------------------------------------ Standard_Dist */
    if (strcmp(loadtype,"h")==0) ok = cipe_cube_write_data(sin,HORIZ_DIST);
    if (strcmp(loadtype,"v")==0) ok = cipe_cube_write_data(sin,VERT_DIST);
    if (strcmp(loadtype,"g")==0) ok = cipe_cube_write_data(sin,GRID_DIST);
```

...cpfilter

```
/*------------------------------------------------------------ End_Standard_Dist */

/*------------------------------------------------------------ Output_Symbol */
    CIPEDATATYPE(sout) = CIPEDATATYPE(sin);
    CIPENS(sout) = CIPENS(sin);
    CIPENL(sout) = CIPENL(sin);
    CIPENB(sout) = CIPENB(sin);
    CIPENDIM(sout) = CIPENDIM(sin);
/* LOADTYPE and others */

    if (strcmp(loadtype,"c")==0) ok = cipe_cube_write_loadmap(sout,HORIZ_DIST);
    if (strcmp(loadtype,"h")==0) ok = cipe_cube_write_loadmap(sout,HORIZ_DIST);
    if (strcmp(loadtype,"v")==0) ok = cipe_cube_write_loadmap(sout,VERT_DIST);
    if (strcmp(loadtype,"g")==0) ok = cipe_cube_write_loadmap(sout,GRID_DIST);
/*------------------------------------------------------------ End_Output_Symbol */

/*------------------------------------------------------------ Load Hypercube-Module */
    cipe_cube_execute_module("nodefilter");
    printf("nodefilter will be started\n");
/*------------------------------------------------------------ End_Load Hypercube-Module */

/*************************************************************************/
/********** HYPERCUBE MODULE INTERFACE SECTION **********/
/*************************************************************************/
/*------------------------------------------------------------ Parameters */
    packet_size = sizeof(int);
    packets = (int *)malloc(nproc*packet_size);

    bcastcp(CIPENAME(sin),sizeof (cipe_sym_name));          /*- input symbol ------* /
    ok=mdumpcp(packets,packet_size,bufmap_receive);
    if (packets[0] == NOK)
        printf("symbol %s  has not been created \n",CIPENAME(sin));

    bcastcp(CIPENAME(sin),sizeof (cipe_sym_name));          /* output symbol ------* /
    ok=mdumpcp(packets,packet_size,bufmap_receive);
    if (packets[0] == NOK)
        printf("symbol %s  has not been created \n",CIPENAME(sin));

    ok=bcastcp(window,sizeof(window));                      /*- window size ------* /
    ok=bcastcp(weight,window[0]*window[1]*sizeof(int));         /*------ weights ------* /
    printf("ok=%d\n",ok);
/*------------------------------------------------------------ End_Parameters */

/*------------------------------------------------------------ Wait */
    ok=mdumpcp(packets,packet_size,bufmap_receive);
    printf("filter has been completed(%d)\n",ok);
/*------------------------------------------------------------ End_Wait */

    free(bufmap_receive);
    free(bufmap_send);
}
```

check_filter

```
check_filter(params,number,value)
struct param *params;
int number;
union {int i; double f; char *s;} value;
{
    int error,whereis;
    error = 0;
    switch(number){
        case 0: whereis=cipe_get_symbol_index(value.s);
                if (whereis == NO_SYMBOL) error = 1;
                if (error) printf("symbol does not exist");
                break;
```

```
            case 1: whereis=cipe_get_symbol_index(value.s);
                    if (whereis != NO_SYMBOL) error = 2;
                    if (error) printf("symbol already exists");
                    break;
            case 2: if (((value.i) % 2) == 0) error = 3;
                    if (error) printf("must be odd number");
                    break;
        }
        if (error != 0) return(-1);
        else return(0);
}
```

struct param params[*number_of_parameters*];
int param_count;
int *error_checking_rout_name*();
int *help_rout_name*();

The last two declarations are only required if the programmer cares to provide the user with help and input parameter error checking. These declarations may be found in the *variable declaration section* labeled *menu related.* In addition, all parameters to be read interactively from a user must also be defined. Such parameters are likely to include references to data sets which are stored as *symbols* within CIPE and referenced by a symbol name. For this reason the application program should request symbol names for input and output. These must be declared by,

cipe_sym_name *input_symbol_name(s), output_symbol_name(s)*;

as shown in the *variable declaration section* labeled *symbol related.* All other interactively read input should be declared in a manner appropriate for the application. In the example application two other such declarations exist,

int window[2];
char loadtype[2];

and they appear in the *variable declaration section* labeled *application related* along with other miscellaneous variables needed by the application. Yamm does not have a character type; all characters must be declared as strings with a minimum length of two; one byte to store the desired character and one byte for the null terminator that must end strings. The **loadtype** above is an example of this type of definition.

With the necessary variable declarations complete, an application programmer may proceed to write the menu mode user interface. This begins with an **if** statement to see if the user is in menu mode or Cli mode, followed by an assignment to **param_count** of the number of **def** statements to be used to read input from the user, the corresponding **def** calls to read the input from the user, and a call to **getpar** to actually create the display screen, and provide help and

input error checking.

```
if (MENU_MODE) {
    param_count = number_of_def_statements;
    def(params,def_stmt_num,"User prompt",param_type,
        variable name,...);
    def(params,...);
    def(params,umber_of_def_statements-1,...);
    if (getpar(params,param_count,error_checking_rout_name,
        help_rout_name);
}
```

In the above example the parameters associated with **def** and **getpar**, which are shown, are required. There are additional parameters that may be given to **def**, and predefined constants that should be used to specify parameter types, or the absence of error checking or help routine names. The sample program uses some of the extensions to the **def** statement. The first **def** statement of the sample program,

```
def(params,0,"input symbol name    ",STRING,input,WID,32,
   LINE,2, REQ,END);
```

reads into the block storage for parameters, **params**, in the first **def** statement, **0**, with the prompt, **"input symbol name"**, a variable of type, **STRING**, into the memory location of, **input**, a value that may take up to the number of columns that follow **WID, 32**, where the prompt will be positioned on the line number following **LINE, 2. REQ** indicates that the value being requested is required not optional, and **END** indicates there are no more parameters or the **def** statement. In the example **getpar** statement,

```
if (getpar(params,param_count,check_filter,NOHELP)) return;
```

the parameter **params** points to an array of parameter definition structures; **param_count**, is the number of **def** statements; the routine to error check the parameters is called **check_filter** (shown at the end of the example program); and, **NOHELP** indicates that no help routine is provided for the parameters.

An error checking routine must have the same name as it was given as an argument to the **getpar** function, and it must have been declared in the *variable declaration section* as mentioned previously. All error checking routines must have three arguments, **params, number, value**, declared as shown in Figure 5.3, which was taken from the sample application program. Within the error checking routine a **switch** statement must be set up. The **cases** of the switch statement, 0-n, correspond to the order of the **def** statements which read in the values being checked; the second parameter to the **def** statement gives its order number. Within each case, checking is done as is appropriate for the value being checked. In some situations it may be possible to use routines provided

```
check_filter(params,number,value)
struct param *params;
int number;
union {int i; double f; char *s;} value;
{
   int error,whereis;
   error = 0;

   switch(number){
      case 0: whereis=cipe_get_symbol_index(value.s);
         if (whereis == NO_SYMBOL) error = 1;
         if (error) error_print("symbol does not exist");
         break;
      case 1: whereis=cipe_get_symbol_index(value.s);
         if (whereis != NO_SYMBOL) error = 2;
         if (error) error_print("symbol already exists");
         break;
      case 2: if (((value.i) % 2) == 0) error = 3;
         if (error) error_print("must be odd number");
         break;
   }
   return(error);
}
```

Figure 5.3 Menu Mode Error Checking Routine

by the executive to assist the programmer in error checking a user's input. One example of such a routine is explained here and others will be explained later in this section. The first **def** statement reads in an input symbol name so the error checking routine under **case 0** checks for the existence of the symbol in the symbol table. It uses the executive routine **cipe_get_symbol_index** which it provides with the symbol name stored in **value.s**, to see if the symbol exists. If it does not, **cipe_get_symbol_index** will return **NO_SYMBOL**, and the user will be given the error message and left in the same menu on the first line to indicate that he/she needs to provide another input symbol. As each parameter is filled in, check_filter is called again with its number in the order of the **def** statements stored in **number**, and the value entered by the user stored in **value**. The assignments to the arguments of an error checking routine and the calling of the error checking routine following each user entry are provided by the menuing package. The second **def** statement read in the output symbol name. An output symbol should not exist, so check_filter under **case 1** again uses **cipe_get_symbol_index**, but this time if the symbol is found it gives the user an error message to say that the symbol already exists. **Case 2** checks to see that the values provided for the filter window size were odd. No error checking is provided for the last **def** statement which asks for the load type. This could be provided by adding a **case 3** to the check_filter switch statement.

Note that each **case** ends with a **break.**

Help may be provided similarly to error checking. The sample program does not provide help. To provide help, a help routine would need to be declared in the variable declarations, and the help routine name would need to be added to the **getpar** statement as its fourth parameter in place of the **NOHELP** constant that was there previously. In the following **getpar** a help routine named **help_filter** has been added.

```
if (getpar(params,param_count,check_filter,help_filter)) return;
```

Then the routine help_filter must be created. All help routines must have two arguments, two of the three required in an error checking routine. A help routine has no need to know the value a user may or may not have entered and so no value is sent to the help routine. The arguments **params** and **number** must be declared as they were for the error checking routine. A switch statement must be set up with cases for each of the **def** statements the programmer desires to provide help. Within the case statements help messages may be printed using **printf.** Yamm will pack the lines and provide new lines. The help routine in Figure 5.4 could be added to the sample application program. For further information on an application program's use of Yamm see the Yamm Programmer's Guide in Appendix G.

5.1.1.2. Creation of a Cli Interface

In Cli mode, the user is not prompted for an application function's input. A Cli user is expected to know the functions being used and their input parameters. When the user inputs a command to execute an application function, the Cli parses the command according to its grammar, identifies it as an application function, and looks up the function name in the *function dictionary.* The function dictionary provides information on the number of parameters that should have been provided to the function, and whether each is an input or an output parameter. The Cli checks for the existence of any symbols which have been entered as parameters to the command. If an input symbol does not exist, the user is provided with an error message that points to the undefined symbol. If an output symbol does exist, the user is also provided with an error message. Values entered as parameters are not associated with a symbol so no error checking is done for them. The Cli passes each parameter of an application function as an argument list of symbol table indices. In order for this to happen, input symbols are retrieved for the user, output symbols are created, and any values are stored as symbols, each under its own temporary name, along with its information on size, type, and values. The Cli command line

```
B = filter(A,3,3,{1,1,1,2,2,2,3,3,3},"c");
```

entered by a user, calls for the execution of a spatial filter function on a data set represented by a symbol A, using a 3 by 3 window with weight values

```
help_filter(params,number)
struct param *params;
int number;
{
   switch (number) {
      case 0:
         printf("A valid input symbol must be associated ");
         printf("with two dimensional image data.");
         break;
      case 1:
         printf("A valid output symbol must not already exist.  ");
         printf("You may see if it does by using List from the");
         printf("Symbol Menu.");
         break;
      case 2:
         printf("Dimensions of the filter window must be ");
         printf("odd numbers.");
         break;
      case 3:
         printf("Horizontal, vertical, grid, and customized ");
         printf("decompositions ");
         printf("have been provided.  Choose one by typing the ");
         printf("first letter of the word.  ");
      break;
   }
}
```

Figure 5.4 Sample Help Routine

specified in the curly brackets, sent to the cube in a customized decomposition, "c". The Cli converts the command line into an argument list which contains six symbol indices to symbols containing A, 3, 3, (1,1,1,2,2,2,3,3,3), "c", and the resultant symbol B.

To simplify the access to the argument list, four macros have been designed:

ARG(n)	–	symbol table index of the n-th argument
ARGPTR(n)	–	pntr to symbol table location of n-th argument
ARGDATA(n)	–	pntr to location of data of n-th argument
RESULTP	–	pntr to symbol table for output/resultant symbol

All the variables a programmer needs to program a Cli interface have already been declared to create the menu interface. A programmer needs only to look at the variable definitions and their functions to determine which of the four macros should be used to assign a value to the variable that represents each of

the user input parameters. For example, for any variables declared to be symbols the macro, **ARGPTR(n)**, should be used to assign a pointer to a symbol table location. For variables that only need to be assigned values, the macro, **ARGDATA(n)**, may be used. **ARGDATA(n)** is a pointer to data and will need to be prefaced with an "*" if its contents are to be accessed. The output symbol variable should be assigned the value of **RESULTP** which contains a pointer to the symbol table entry of the output symbol. The macro definitions assume the existence of only one output symbol; for future applications this will need to be modified. The lines,

```
sin = ARGPTR(0);
window[0] = *(int *)ARGDATA(1);
window[1] = *(int *)ARGDATA(2);
```

begin by taking the pointer to the symbol table location of the first argument, 0, and assigning it to **sin**, which is of type **cipe_symbol**. The second line takes the contents of the data location of the second argument in the argument list, 1, and assigns it to **window[0]**. The third line takes the contents of the data location of the third argument, 2, and assigns it to **window[1]**. See the *user interface section* labeled *Cli* for a complete Cli interface.

5.1.2. Host System Resident Monitor Interface

An application program must interface with the host system resident monitor to create symbols (**cipe_creat_symbol**), access symbols (**cipe_get_symbol**), distribute data to the hypercube (**cipe_cube_write_data**), activate its counterpart hypercube module (**cipe_cube_execute_module**), and to perform a variety of other activities. Interfacing to the host resident monitor is done through functions CIPE provides for the tasks it expects a programmer to need to do. In this manner the job of the programmer is simplified and CIPE applications appear more uniform. A complete list of routines provided by CIPE may be found in Appendix C which is subdivided by routine functionality.

The first time a host module is likely to need to interface with the host resident monitor is to retrieve input symbol(s) and to create output symbol(s). In Cli mode these functions happen transparently, as explained previously, so that Cli needs only to pass symbol table indices in its argument list. In menu mode it is necessary to retrieve the input symbol(s) and create the output symbol(s) from their symbol names. A symbol is retrieved through the use of the command **cipe_get_symbol**, and created through the use of the command **cipe_creat_symbol**. In order to retrieve a symbol a variable must have been declared to store the symbol. The routine which retrieves symbols **cipe_get_symbol** must also be declared to return a pointer to a symbol structure. In order to create a symbol, variables must have been declared to store the symbol and a symbol index. These declarations look like,

```
struct cipe_symbol *sin, *sout, *cipe_get_symbol();
int sindex;
```

and may be found in the *variable declaration section* labeled *symbol related.* The commands are used by,

```
sin = cipe_get_symbol(input);
ok = cipe_creat_symbol(output,&sout,&sindex);
```

and may be found in the *CIPE host resident monitor interface section* labeled *symbol retrieval*, which gives a completed menu interface. In the future, Cli and menu modes will function consistently for retrieving and creating symbols.

Once an application program has completed its collection of information from the user, and has retrieved or created all necessary symbols, the program is ready to distribute data to the hypercube. If the programmer desires to use one of the standard distribution types, he/she may do so simply by using the routine, **cipe_cube_write_data**, giving as its arguments only an input symbol and a distribution type keyword. Valid keywords for standard distributions are:

BCAST_DIST	-	*sends the entire data set to each node*
HORIZ_DIST	-	*creates a horizontal distribution*
GRID_DIST	-	*creates a grid distribution*
VERT_DIST	-	*creates a vertical distribution*

The keywords may be used with the executive routine by the statement,

ok = cipe_cube_write_data(*input_symbol,distribution_type_keyword***);**

as may be found in the *host resident monitor interface section* labeled *standard_dist.*

The sample filter application allows the user to choose one of four data decompositions, but to properly filter the image requires something other than a standard distribution. This is provided by using the same command, **cipe_cube_write_data**, with a different keyword, **CUSTOM_DIST**, which allows the programmer to create any kind of a data decomposition desired. To use the **CUSTOM_DIST** keyword a programmer must first create a load map for the input symbol. In the case of standard distributions, **cipe_cube_write_data** completes the load map for the programmer. To create a load map a programmer must access the map's storage location within the symbol. The attributes of a symbol can be cumbersome to reference. To assist the programmer in doing so, CIPE provides the following macro definitions.

CIPEDATA(*symbol*)	-	**Pointer to the start of the data**
CIPENAME(*symbol*)	-	**Name associated with the symbol**
CIPEFILE(*symbol*)	-	**File name associated with symbol**
CIPENDIM(*symbol*)	-	**Number of dimensions in the data**
CIPESB(*symbol*)	-	**Starting band of the data**
CIPENB(*symbol*)	-	**Number of bands of the data**
CIPESL(*symbol*)	-	**Starting line of the data**
CIPENL(*symbol*)	-	**Number of lines of the data**
CIPESS(*symbol*)	-	**Starting sample of the data**
CIPENS(*symbol*)	-	**Number of samples of the data**
CIPEDATATYPE(*symbol*)	-	**Type of each data element**
CIPELOADTYPE(*symbol*)	-	**Data distribution type**
CIPELOADMAP(*symbol*)	-	**Pointer to the load map of the data**
CIPEELEMENTSIZE(*symbol*)	-	**Size of an element of the data**
CIPENUMELEMENTS(*symbol*)	-	**Number of elements in the data**
CIPEDATASIZE(*symbol*)	-	**Total size of the data**

The structure definitions associated with these macros may be found in Appendix I under *symbol.h*. To create a load map a programmer needs to use the **CIPELOADMAP(symbol)** macro. CIPELOADMAP stores a pointer to an array containing the starting line, starting sample, starting band, number of lines, number of samples, and number of bands that each node is to receive when data is downloaded. The macros CIPESB, CIPENB, CIPESL, CIPENL, CIPESS, and CIPENS, store similar information for the entire data set as it exists in the host, which is not to be confused with the data pointed to by CIPELOADMAP which stores the information for the individual hypercube nodes. To make assignments to CIPELOADMAP a loop similar to the following must be created.

```
for (i=0; i<number_of_cube_nodes; i++) {
   CIPELOADMAP(input_symbol)[i].sl  = starting line for node i;
   CIPELOADMAP(input_symbol)[i].nl  = number of lines for node i;
   CIPELOADMAP(input_symbol)[i].ss  = starting sample for node i;
   CIPELOADMAP(input_symbol)[i].ns  = number of samples for node i;
   CIPELOADMAP(input_symbol)[i].sb  = starting band for node i;
   CIPELOADMAP(input_symbol)[i].nb  = number of bands for node i;
}
```

Once the load map has been created the data may be downloaded using the command:

```
ok = cipe_cube_write_data(input_symbol,CUSTOM_DIST);
```

For the the sample application a horizontal decomposition with overlapping data regions is desired. The appropriate load map was created in the *host resident monitor interface section* labeled *CUSTOM_DIST*.

In addition to providing for the download of input data, a programmer must provide the means of uploading output data before invoking the host's corresponding node module. To accomplish this the output symbol's attributes must be assigned values. Typically, but not always, this involves assigning the output symbol's attributes the same values as for the input symbol, since one would expect the output data to be of the same size, dimension, *etc.* and only differing in the actual contents of the data. This may be done by:

CIPEDATATYPE(*symbol_out*) = **CIPEDATATYPE**(*symbol_in*);

Similar statements should be made for other attributes. The load map of an output symbol may be specified for an application program in the same manner as for an input symbol. Again a programmer may choose a standard or customized distribution. While the input symbol's loadmap implies the data area that each node must process, the output symbol's loadmap implies the area of the output that each node is responsible to generate. A complete output symbol setup is shown in the *host resident monitor interface section* labeled *output_symbol.*

5.1.3. Hypercube Module Interface

An application's host module may request activation of its corresponding hypercube module with the following command.

cipe_cube_execute_module("*node_module_name*");

This command interacts with the host resident monitor. The host resident monitor is only capable of dealing with one command at a time. For this reason, prior to the activation of the hypercube module, the input data must be distributed to the cube, the output symbol load map must have been established, and any other commands interacting with the host resident monitor must be complete. This command may be found in the sample program in the *host resident monitor interface section* labeled *load hypercube-module.*

Once the hypercube module is activated, it waits for the parameters from the host module. In the future, CIPE will provide a function to pass parameters between the host and hypercube modules. At the current time, parameters must be passed using the hypercube's Crystalline Operating System (CrOS) functions. In order to use CrOS functions, the file, *cros.h*, must be included. The parameter passing procedure for most applications may be accomplished with two CrOS functions, **bcastcp** and **mdumpcp**. The *bcastcp* is for broadcasting a parameter value to all of the hypercube nodes and *mdumpcp* is for receiving an acknowledgment from each node. The CrOS function calls of the host module must be synchronized with the corresponding CrOS function calls (**bcastelt** and **dumpelt**) of the hypercube module in order for the parameters to be transferred properly. The hypercube side CrOS calls will be discussed in the hypercube module programming section. The CrOS commands **bcastcp** and **mdumpcp** are used as follows.

```
ok = bcastcp(pointer_to_data_to_be_sent,number_of_bytes_to_send);
if (ok == -1) {
   printf("cpfilter: error in bcastcp sending x to nodes\n");
   return(-1);
}
ok = mdumpcp(pointer_to_rcv_data,max_bytes_per_node,
   pointer_to_array_bytes_rcvd);
for (i=0; i<number_of_processors; i++)
   if (packets[i] == NOK)
      printf("Error in node %d completing task x\n",i);
```

If a programmer desires to pass distinct parameters to each node, or to communicate between the host and node modules for other functions, additional information on these and other hypercube commands may be found in the *Mark III Hypercube Programmer's Manual.*

The parameter passing example can be found in the sample filter program in the *hypercube module interface section* labeled *parameters*. The parameter acknowledgment handshaking is not a mandatory requirement but is recommended for possible error detection prior to applying the data processing function.

After the host program provides for the input parameters to be sent to the hypercube, it waits for the hypercube resident module to complete the data processing. If the programmer has chosen for the nodes to acknowledge reaching certain check points, the host module will need to provide for the receipt of the messages sent back. The rest of the work is left to the hypercube module.

One host system resident module may activate more than one hypercube resident module sequentially, if desired. For example, a map projection program may activate a hypercube module for geometric transformation and then another module for resampling. A hypercube module is better designed to perform a simple function so that it can be used by various host modules. CIPE intends to provide a large set of hypercube data processing modules for image processing applications so that an application programmer may need to write only host side modules calling the needed hypercube modules.

5.2. HYPERCUBE RESIDENT PROGRAMMING

The hypercube resident application modules consist of the computationally intensive part of the application programs. Similar to the host resident application modules, all hypercube resident modules have a common set of program procedures. Each hypercube module must interface with the host module to receive input parameters, must interact with the hypercube resident monitor for accessing input and output symbol structures, and must process data. All hypercube resident modules must include *elt_mon.h* and *elt_symbol.h* to provide some necessary variable and structure definitions.

Before processing any data, the hypercube module needs to read the necessary input parameters. The host module is using CrOS communication routines to send the parameters to the nodes and therefore the hypercube module needs to use the CrOS communication routines that correspond to those in the host. To utuilize CrOS communication routines the hypercube resident module must include *cros.h.* It has already been explained that it will be most common for the host to broadcast the same parameters to all the nodes. The command in the hypercube that corresponds to the host's broadcast command is **bcastelt.** The programmer may like for the hypercube module to acknowledge its progress to the host. The hypercube module may do this by periodically sending back information to the host. Each node may send distinct information back to the host by the use of the CrOS routine **dumpelt.** The commands **bcastelt** and **dumpelt** are used as follows.

```
ok = bcastelt(pointer_to_rcv_info,max_bytes_to_rcv);
if (ok == -1) {
   printf("Error occurred in bcastelt receiving info x0);
   return(-1);
}
ok = dumpelt(pointer_to_info_to_send,bytes_to_send);
if (ok == -1) {
   printf("Error occurred in dumpelt sending info x0);
   return(-1);
}
```

If the **bcastelt** or **dumpelt** commands fail they will return a "-1". The first parameter in each of the routines must be a pointer and the second must be an integer. Examples of the use of these routines may be found in Figure 5.5 in the *parameters section.* All future references to an application program will refer to the sample hypercube resident module in Figure 5.5.

The host module of an application program needs to pass input and output symbol names to the hypercube module. These will be passed as for all parameters as previously discussed. The entire input symbols and the output symbol attributes have already been passed and stored in the hypercube, but the hypercube module of the application needs to know the symbol names in order to retrieve them. Symbols are retrieved by using provided routines which interface with the hypercube resident monitor. Symbols may be retrieved using the routine, **elt_get_symbol.** The routine **elt_get_symbol** takes the name of a symbol and a constant indicating whether it is a local or a global symbol, and it returns a pointer to the symbol's position in the symbol table. Symbols created with the provided **creat_symbol** routine are global and are intended to be used to share data between application programs. Within the node module of the application program, the pointer to be used to reference global symbols must be declared globally as well. See the *Global Symbols section* of the sample node module.

```
#ifndef lint
static char sccsid[]="%W% %G%";
#endif lint

/* %M% version %I%, %G% */
#include <stdio.h>
#include <ctype.h>
#include <cros.h>
#include <errno.h>
#include "elt_mon.h"
#include "elt_symbol.h"

#define NOK 0
#define OK 1

/*------------------------------------------------------------ Global Variables */
int packet;
int packet_size;
extern doc,nproc,procnum;

/*------------------------------------------------------------ Global Symbols */
struct elt_symbol *sin, *sout,  *elt_get_symbol();

nodefilter()                                                          nodefilter
{
    elt_name input,output;
    int nlw,nsw,hnsw,*weight;
    int i,j,k1,k2,ioff,ioff1,ok,window[2];
    float *fweight,sum;
    unsigned char *image_data,*creat_global();
    register unsigned char *p1;

/*------------------------------------------------------------ Parameters */
    packet_size = sizeof(packet);

    bcastelt(input,sizeof(input));                          /*---------- input symbol ----------*/
    sin = elt_get_symbol(input,GLOBALSYM);
    if (sin == NULLSYM) packet = NOK;
    else packet = OK;
    ok=dumpelt(&packet,packet_size);
    image_data = DATA(sin);

    bcastelt(output,sizeof(output));                        /*---------- output symbol ----------*/
    sout = elt_get_symbol(output,GLOBALSYM);
    if (sout == NULLSYM) packet = NOK;
    else packet = OK;
    ok=dumpelt(&packet,packet_size);

    bcastelt(window,sizeof(window));                        /*---------- window size ----------*/
    nlw = window[0];
    nsw = window[1];
    printf("window size (%d %d)\n",nlw,nsw);
    hnsw = nsw /2;
    weight = (int *)malloc(nlw*nsw*sizeof(int));          /*---------- weights ----------*/
    ok = bcastelt(weight,nlw*nsw*sizeof(int));
    printf("weight received (%d)\n",ok);
/*------------------------------------------------------------ End_Parameters */

/*------------------------------------------------------------ Filter Process */
/* make the sum of the weight matrix to be 1 - normalize the weight matrix    */
    fweight = (float *)malloc(nlw*nsw*sizeof(float));
    sum =0.0;
    for (k1=0; k1<nlw*nsw; k1++) sum += (float)weight[k1];
    for (k1=0; k1<nlw*nsw; k1++) fweight[k1] = (float)weight[k1] /sum;
```

Figure 5.5 Spatial Filter Node Program

...nodefilter

```
    for (k1=0; k1<nlw; k1++)
        for (k2=0; k2<nsw; k2++)
            printf("weight(%d,%d) = %f\n",k1,k2,fweight[k1*nsw+k2]);

/* apply filter */
    printf("filter starts (%d %d),(%x %x),(%d %d)\n",
        NL(sin),NS(sin),DATA(sin),DATA(sout),NL(sout),NS(sout));
    ioff = 0;
    for (i=0; i<NL(sin)-nlw; i++) {
        p1 = DATA(sout)+i*NS(sout)+hnsw;
        for (j=0; j<NS(sin)-nsw; j++) {
            sum = 0.0;
            ioff1 = ioff+j;
            for (k1=0; k1<nlw; k1++) {
                for (k2=0; k2<nsw; k2++)
                    sum += (float)image_data[ioff1+k2] * fweight[k1*nsw+k2];
                ioff1 += NS(sin);
            }
            if (sum < 0.0) sum = 0.0;                    /* under saturation */
            if (sum > 255.0) sum= 255.0;                 /* over saturation  */
            *p1++ = sum;
        }
        ioff += NS(sin);
    }
    printf("filter is completed\n");

    free(weight);
    free(fweight);
    packet = OK;
    ok = dumpelt(&packet,sizeof(int));
/*------------------------------------------------------------ End_Filter */
}
```

Once all the parameters have been read and the symbols retrieved the application program may process the data. The data processing part of the application program may be written similarly to the way it would be for a sequential program. The algorithm is likely to be the same with only the bounds on the data set changing. The hypercube resident module may provide status reports to the host resident module as desired by use of the CrOS **dumpelt** routine.

Upon completion of the node module the hypercube resident monitor will be ready to receive another command. The application program needs to take no action to return the processed data to the host. The data will remain resident in the hypercube for processing by another application until the user chooses to retrieve the symbol to the host for display or storage, or to delete the symbol.

5.3. INTEGRATING APPLICATION PROGRAMS INTO CIPE

Each of the application program's two modules, the host resident module and the hypercube resident module, must be appropriately linked with CIPE in order to run. This happens through the use of **Makefiles** like the samples provided. Figure 5.6 provides a **Makefile** for a host resident application module.

```
CC=ccc
CIPEDIR=/spacely/ufs/cipe
CIPE = $(CIPEDIR)/cipe
CFLAGS= -g -I$(CIPEDIR)/include
APPLS= host_resident_appl_module_name
LINK= $(CIPEDIR)/appl/cp/CPLINK

host_resident_appl_module_name: host_resident_appl_module_name.o \
        $(CIPE)
    $(LINK) $(CIPE) $@
```

Figure 5.6 Host Resident Application Module Makefile

A programmer needs only to place his/her own host resident application module name where it says *host_resident_appl_module_name*. This links host modules with CIPE and the host resident monitor. Figure 5.7 provides a makefile for a hypercube resident application module.

```
CC=/jane/ufs/cube/bin/cc68
CIPEDIR=/spacely/ufs/cipe
CIPE = $(CIPEDIR)/cipe
CFLAGS= -g -I$(CIPEDIR)/include
APPLS= cube_resident_appl_module_name
ELT_MONITOR= $(CIPEDIR)/cube/elt_monitor/elt_mon
LINK= $(CIPEDIR)/appl/elt/ELTLINK

cube_resident_appl_module_name: cube_resident_appl_module_name.o \
        $(ELT_MONITOR)
    $(LINK) $(ELT_MONITOR) $@
```

Figure 5.7 Hypercube Resident Application Module Makefile

A programmer needs only to place his/her own hypercube resident application module where it says *cube_resident_appl_module_name*. This links cube resident modules with the hypercube resident monitor.

Application programs linked as described may be called within CIPE from menu or Cli mode. They may be called from menu mode by going to the **appl** menu option which allows the user to type in the application function name and the requested application program will be loaded. In order for a user to call his/her own application function from Cli mode, an entry for the application must be added to the *function dictionary*. All applications for use by CIPE are listed in a *function dictionary*. Each entry of the *function dictionary* has one of the following forms, depending on whether or not the application returns a value:

FUNCTION *name* (*argument-list*)
RETURNS (*argument-list*)
HELP "*help message*"
PATHNAME *location-list*

PROCEDURE *name* (*argument-list*)
PATHNAME *location-list*

where an *argument-list*, possibly empty, is structured as

[*access-type*] [*value-type*] *prompt-message*

for

access-type **INPUT** *or* **OUTPUT**
value-type **INTEGER, FLOAT, DOUBLE, STRING,** *or* **BOOLEAN**
prompt-message "*string*"

and a *location-list* is structured as

pathname **USES** *system*

Two examples follow.

```
function filter(input "In",
   input "Box width",
   input "Box height",
   input "Wgts",
   input "Loadtype")
   returns "Out" pathname "/ufs/cipe/appl/cp/cpfilter"

procedure cleardisplay( )
   pathname "/ufs/cipe/appl/cp/cleardisplay"
```

5.4. APPLICATIONS WITHOUT A COPROCESSOR

Application programming in CIPE without the use of a coprocessor is much simpler as there is only a host resident module, and therefore no data distribution or parameter passing to the coprocessor is necessary. Figure 5.8 is a filter program which runs in CIPE without the use of a coprocessor. Figure 5.2, discussed previously, was an example of a host resident module of a filter program which used the JPL/Caltech Mark III as a coprocessor. Close comparison of Figure 5.2 and Figure 5.8 will demonstrate the similarities and differences between application programming in CIPE with and without a coprocessor. These similarities and differences will be presented here.

The *variable declaration section* of Figures 5.2 and 5.8 are similar. The sections labeled *menu related* and *symbol related* are identical in the two examples. Figure 5.8 lacks any hypercube-related variables because it is not intended to use the hypercube or any other coprocessor. Additional variables are found in the area labeled *application related* in order to support the data processing that will now occur in the host instead of in a coprocessor.

The *user interface section* of the two examples is similar, as well. Without the hypercube, the filter program does not need to request a data load type of the programmer. Therefore, the *user interface section* labeled *menu* lacks the **def** statement which reads the load type, and the count of **def** statements assigned to **param_count** is one less. The lack of a load type means that the *user*

```
#ifndef lint
static char sccsid[]="%W% %G%";
#endif lint

/* %M% version %I%, %G% */
#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <cros.h>
#include "cipeappl.h"

seqfilter()                                                    seqfilter
{

/************************************************************************/
/****************** VARIABLE DECLARATION SECTION ****************/
/************************************************************************/
/*------------------------------------------------------------ Menu Related */
    struct param *params;
    int param_count;
    int check_filter();

/*------------------------------------------------------------ Symbol Related */
    cipe_sym_name input,output;
    struct cipe_symbol *sin, *sout, *cipe_get_symbol();
    int sindex;

/*------------------------------------------------------------ Application Related */
    int nlw, nsw, hnsw, window[2], *weight, *weightptr;
    int k1,k2,i,j,ioff,ioff1;
    int ok;
    float sum,*fweight;
    unsigned char *image_data;
    unsigned char register *p1;


/************************************************************************/
/********************** USER INTERFACE SECTION **********************/
/************************************************************************/
/*------------------------------------------------------------ Menu */
    if (MENU_MODE) {
        param_count = 3;
        params = (struct param *)malloc(sizeof(struct param) * param_count);

        def(params,0,"input symbol name          ",STRING,input,WID,32,LINE,2,
        REQ,END);
        def(params,1,"output symbol name         ",STRING,output,WID,32,LINE,3,
        REQ,END);
        def(params,2,"filter window(nlw,nsw) ",INT,window,INCR,4,WID,10,
        LINE,4,DUP,2,REQ,END);
        if (getpar(params,param_count,check_filter,NOHELP)) return;
        free(params);

        params = (struct param *)malloc(sizeof(struct param)*window[0]*window[1]);
        weight = (int *)malloc(window[0]*window[1]*sizeof(int));

        for (i=0; i<window[1]; i++) {
            def(params,i,"weight values ",INT,weight+i,
            INCR,sizeof(int)*window[0],LINE,2,
            ENTRYLINE,3,ENTRYCOL,i*10,DUP,window[0],GROUP,0,REQ,END);
        }
        if (getpar(params,window[1],NOECHK,NOHELP)) return;
    }
```

Figure 5.8 Spatial Filter Program Without a Coprocessor

...seqfilter

```
/*------------------------------------------------------------------ End_Menu */

/*------------------------------------------------------------------ Cli */
    else {
          sin = ARGPTR(0);
          window[0] = *(int *)ARGDATA(1);
          window[1] = *(int *)ARGDATA(2);
          weight = (int *)malloc(window[0]*window[1]*sizeof(int));
          weightptr = (int *)ARGDATA(3);
          for (i=0; i<window[0]*window[1]; i++)
               weight[i] = weightptr[i];
          sout = RESULTP;
    }
/*------------------------------------------------------------------ End_Cli */

/*****************************************************************/
/******** CIPE HOST RESIDENT MONITOR INTERFACE ********/
/*****************************************************************/
/*------------------------------------------------------------------ Symbol Retrieval */
    if (MENU_MODE) {
          sin = cipe_get_symbol(input);
          ok = cipe_creat_symbol(output,&sout,&sindex);
    }

    cipe_host_write_data(sin);
/*------------------------------------------------------------------ End_Symbol Retrieval */

/*------------------------------------------------------------------ Output_Symbol */
    CIPEDATATYPE(sout) = CIPEDATATYPE(sin);
    CIPENS(sout) = CIPENS(sin);
    CIPENL(sout) = CIPENL(sin);
    CIPENB(sout) = CIPENB(sin);
    CIPENDIM(sout) = CIPENDIM(sin);
    CIPEDATA(sout) = (unsigned char *)malloc(CIPEDATASIZE(sout));
/*------------------------------------------------------------------ End_Output_Symbol */

/*****************************************************************************/
/************************** DATA PROCESSING **************************/
/*****************************************************************************/
/*------------------------------------------------------------------ Filter Process */

    nlw = window[0];
    nsw = window[1];
    hnsw = nsw /2;

/* make the sum of the weight matrix to be 1 - normalize the weight matrix */
    fweight = (float *)malloc(nlw*nsw*sizeof(float));
    sum =0.0;
    for (k1=0; k1<nlw*nsw; k1++) sum += (float)weight[k1];
    for (k1=0; k1<nlw*nsw; k1++) fweight[k1] = (float)weight[k1] /sum;
    for (k1=0; k1<nlw; k1++)
          for (k2=0; k2<nsw; k2++)
               printf("weight(%d,%d) = %f\n",k1,k2,fweight[k1*nsw+k2]);

/* apply filter */
    image_data = CIPEDATA(sin);
    ioff = 0;
    for (i=0; i<CIPENL(sin)-nlw; i++) {
          p1 = CIPEDATA(sout)+i*CIPENS(sout)+hnsw;
          for (j=0; j<CIPENS(sin)-nsw; j++) {
               sum = 0.0;
               ioff1 = ioff+j;
               for (k1=0; k1<nlw; k1++) {
                    for (k2=0; k2<nsw; k2++)
```

...seqfilter

```
                    sum += (float)image_data[ioff1+k2] * fweight[k1*nsw+k2];
                ioff1 += CIPENS(sin);
            }
            if (sum < 0.0) sum = 0.0;                    /* under saturation */
            if (sum > 255.0) sum= 255.0;                 /* over saturation  */
            *p1++ = sum;
        }
        ioff += CIPENS(sin);
        printf("ioff= %d,   p1 = 0x%x\n",ioff,p1);
    }
    printf("filter is completed\n");

    free(weight);
    free(fweight);
/*------------------------------------------------------------------ End_Filter */
}
```

check_filter

```
check_filter(params,number,value)
struct param *params;
int number;
union {int i; double f; char *s;} value;
{
    int error,whereis;
    error = 0;
    switch(number){
    case 0: whereis=cipe_get_symbol_index(value.s);
            if (whereis == NO_SYMBOL) error = 1;
            if (error) printf("symbol does not exist");
            break;
    case 1: whereis=cipe_get_symbol_index(value.s);
            if (whereis != NO_SYMBOL) error = 2;
            if (error) printf("symbol already exists");
            break;
    case 2: if (((value.i) % 2) == 0) error = 3;
            if (error) printf("must be odd number");
            break;
    }
    if (error != 0) return(-1);
    else return(0);
}
```

interface section labeled *Cli* has one less argument in its argument list, and it therefore lacks the line that previously read the load type from the argument list.

The *cipe host resident monitor interface section* no longer provides for distributing data to the hypercube. Instead, the function **cipe_host_write_data** is called to make sure that the data of the input symbol is resident in the host system memory. Prior to the function call the data of the input symbol may be in the file, in the coprocessor, or already in the host system memory, depending on previous functions applied to the symbol. An example of the function call may be found in the area labeled *retrieve_symbol*. The *cipe host resident monitor interface section* also still provides for assigning the attributes of the output data symbol which is found in the area labeled *output_symbol*. Previously, when the filter program used the hypercube, no data space was allocated for the output symbol in the host because the data resided in the hypercube. When outside of the application program the user chose to read the data back from the hypercube the space would have been created. Now it is necessary to allocate space in the host for the output symbol because that is where the data will reside. The last line of the section labeled *output symbol* provides for this.

```
CIPEDATA(sout) = malloc(CIPEDATASIZE(sout));
```

The remaining portion of Figure 5.8 provides for the actual filtering of the data. Previously, in Figure 5.2, a node module would have been activated and the host module would have passed its parameters and waited for execution. Now the same filtering algorithm used in the nodes resides in the host and executes on the entire image instead of the subset each node previously operated on.

6. FUTURE DIRECTION

The previous five sections described the features and implementation details of CIPE with respect to the user environment, programming environment, and internal management of data and image processing functions. The user environment emphasizes friendly access to provided functions, and the programming environment provides system architecture-independent subroutine layers for user interface, file I/O, hypercube utilization, and display device manipulation. A symbol-oriented data management and an incremental program loading mechanism are implemented for more efficient utilization of the concurrent system, and to overcome the system I/O bottleneck common in a high rate data processing environment. The future direction of CIPE will be divided in three parts, based on the three distinctive characteristics of CIPE: as an interactive image processing executive, as a concurrent system executive, and as a high rate data processing executive.

The future CIPE as an interactive image processing executive will address a new environment where a user and programmer may become one. The fundamental desire of every user is to express his/her thought process and to experiment with ideas without having to go through the cumbersome transformation process of programming. Such an environment is generally referred to as an automatic programming environment. Computer scientists have approached automatic programming via development of high level languages which reduce the effort of transforming a human logical process into a computer's machine instructions. Although high level languages significantly simplify programming procedures, it is far from achieving true automatic programming. Automatic programming in a generic sense may be an illusion, since one generic language cannot possibly understand nor provide all users' needs. However, within a well-defined user community, such an environment may be achievable. The scoping of the user community and characteristics of users are essential for a successful implementation of a proper user environment.

The future goal of CIPE is to provide such an automatic programming environment for an image processing user community with scientifically oriented users. The goal will be pursued in three major directions: a user interface, an application tool box, and higher level image processing language development. The future user interface will be directed toward developing various expressional tools which can compensate the limitations of typing from a terminal. The tools will utilize more advanced window-based terminals, display devices, and various input sources. The development of an application tool box will be pursued via analyzing image processing algorithms to extract a set of basic image processing operators (primitives). The complete set of operators will allow any image processing to be performed using a proper combination of the operators. Finally, an image processing language which allows manipulation of the image processing operators will be studied and developed. The language will allow usage in an interpretive mode as well as in a programming mode.

The future CIPE as a concurrent system executive will incorporate various concurrent architectures and provide a more advanced programming environment for such architectures. An automatic parallel compiler will be researched to remove the requirement of parallel programming for a specific architecture. The compiler will not be general purpose, but will be oriented toward an image processing programming environment where a large data set is distributed among multiple nodes in a concurrent system for execution of a same function.

The future CIPE as a high rate data processing executive will provide necessary testbed functions for EOS era. Such functions will include sensor simulation, benchmarking of systematic processing procedures, and/or simulation of various architectural concepts for on-board data processing and analysis.

Appendix A – CIPE FILES

Directory	Filename	Link Module
include	cipe.h	
	symbol.h	
	funclib.h	
	image_hdr.h	
	menus.h	
	opcodes.h	
	builtin.h	
	cp_mon.h	
	elt_mon.h	
	elt_data.h	
	elt_symbol.h	
main	cipe.c	libmain.a
	dictionlex.l	
	y.tab.h	
	diction.y	
	lex.yy.c	
	parser2.sed	
	symbol.c	
	funclib.c	
cli	codegen.c	libcli.a
	ops.c	
	y.tab.c	
	lex.yy.c	
	parser.c	
menu	cipemenu.c	libmenu.a
	dispmenu.c	
	setmenu.c	
	applmenu.c	
exec	docli.c	libdo.a
	doclisubs.c	
	doset.c	
	dodisp.c	
	doplot.c	
	doplot3d.c	
	doplotsubs.c	
	doio.c	
	dofile.c	
	dobltin.c	
	dobltinc.c	
	docall.c	
	doload.c	
	docube.c	
	dosymbol.c	

Directory	Filename	Link Module
cp_monitor	cp_mon.c	cp_mon.o
	cp_data.c cp_data.o	
elt_monitor	elt_mon.c	elt_mon
	elt_symbol.c	
	elt_module.c	
	elt_data.c	
	elt_bltin.c	
appl/cp	cpfilter.c	
	cpmed	
	cpgeom.c	
	cpzoom.c	
	cphist.c	
	cpproj.c	
	cprender.c	
	cpfft2.c	
appl/elt	nodefilter.c	
	nodemed	
	nodegeom.c	
	nodezoom.c	
	nodehist.c	
	nodeproj.c	
	noderender.c	
	nodefft2.c	

SUPPORTING LIBRARIES

/usr/local/lib	
	ydlib.a
	xdlib.a
	ivasx.a
	ivas.a
	xanth.a
	yamm.a

Appendix B – COMMAND LINE INTERPRETER SUBROUTINES

Subroutine: **struct code_seg_def *cipe_init_code_seg(name,workspace)**

File: codegen.c

Input: **char *name;**
unsigned char workspace;

Output: None

Function: **Cipe_init_code_seg** creates a new code segment, complete with symbol table and function dictionary. This code segment then becomes the current code segment; the previous code segment is saved. The function returns a pointer to the new code segment.

Suggested Modification:
None

Subroutine: **cipe_restore_code_seg()**

File: codegen.c

Input: None

Output: None

Function: Returns to the previous (parent) code segment. The current code segment is left intact. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_arith_op(op,arg1,arg2,result)**

File: codegen.c

Input: **unsigned int op;**
symbolnum arg1,arg2,result;

Output: None

Function: **Cipe_arith_op** generates an arithmetic instruction to be executed by the Command Line Interpreter either immediately (as in the case of a request to print out the sum of two numbers), or in the future execution of a user-defined workspace or function. A temporary variable is created in the code segment symbol table, if necessary, and the instruction is saved. The function returns the symbol number (symbol table index) of the result variable or **NO_SYMBOL** on error.

Suggested Modification:
None

Subroutine: **cipe_assign(left_side,right_side)**

File: codegen.c

Input: **symbolnum left_side,right_side;**

Output: None

Function: **Cipe_assign** handles assignments requested interactively or in the execution of a user-defined workspace or function. Assignments are optimized away by the function, if possible. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_index(var,index_list)**

File: codegen.c

Input: **symbolnum var;**
struct indexlist_def *index_list;

Output: None

Function: **Cipe_index** generates a subscript instruction to be executed by the Command Line Interpreter either immediately (as in the case of a request to display a small region of an image), or in the future execution of a user-defined workspace or function. The function returns the symbol number (symbol table index) of the result variable or **NO_SYMBOL** on error.

Suggested Modification:
None

Subroutine: **cipe_call(func,expr_list)**

File: codegen.c

Input: **functionnum func;**
struct exprlist_def *expr_list;

Output: None

Function: **Cipe_call** generates a request to call the specified function to be executed by the Command Line Interpreter either immediately (as in the case of an interactive function call), or in the future execution of a user-defined workspace or function. The function returns the symbol number (symbol table index) of the result variable or **NO_SYMBOL** on error.

Suggested Modification:
None

Subroutine: **cipe_br(label_loc)**

File: codegen.c

Input: **struct instruction *label_loc;**

Output: None

Function: **Cipe_br** takes an instruction pointer (label location) and generates an instruction to branch to that location. This is normally called in the compilation of a conditional or looping construct. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **struct instruction *cipe_add_br_placeholder()**

File: codegen.c

Input: None

Output: None

Function: **Cipe_add_br_placeholder** generates a placeholder instruction in the compilation of a looping or conditional construct in a user-defined workspace or function. The placeholder instruction is initially an NOP, which is normally replaced later in code generation. The function returns a pointer to the placeholder instruction.

Suggested Modification:
None

Subroutine: **cipe_patch_br(br_loc,label_loc)**

File: codegen.c

Input: **struct instruction *label_loc;**

Output: **struct instruction *br_loc;**

Function: **Cipe_patch_br** patches a previously compiled unconditional branch instruction using new information. The absolute location in **label_loc** is converted to a code-segment-relative location and stored in the specified instruction, together with a branching opcode in case the previous opcode was a placeholder. This function is normally used in the compilation of conditional and looping constructs. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_patch_brz(br_loc,label_loc,expr1)**

File: codegen.c

Input: **struct instruction *label_loc;**
symbolnum expr1;

Output: **struct instruction *br_loc;**

Function: **Cipe_patch_brz** patches a previously compiled branch-on-zero instruction using new information. The absolute location in **label_loc** is converted to a code-segment-relative location and stored in the specified instruction, together with a branch-on-zero opcode in case the previous opcode was a placeholder. The expression to use in the condition is also replaced. This function is normally used in the compilation of conditional and looping constructs. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_patch_brfor(br_loc,label_loc,expr1,expr2,step)**

File: codegen.c

Input: **struct instruction *br_loc,*label_loc;**
symbolnum expr1,expr2,step;

Output: None

Function: **Cipe_patch_brfor** patches a previously compiled branch-in-for instruction using new information. The absolute location in **label_loc** is converted to a code-segment-relative location and stored in the specified instruction, together with a branch-in-for opcode in case the previous opcode was a placeholder. The expressions denoting the limits of the loop and the step size to use are also replaced. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_set_attr(attr,value)**

File: codegen.c

Input: **unsigned int attr;**
symbolnum value;

Output: None

Function: **Cipe_set_attr** generates an instruction to set a particular system attribute. This instruction may either be executed immediately (as in the case of an interactive processor allocation), or in the future execution of a user-defined workspace or function. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_wscmd(cmd_code,ws)**

File: codegen.c

Input: **unsigned int cmd_code;**
symbolnum ws;

Output: None

Function: **Cipe_wscmd** generates an instruction to reference (load, save, edit) a workspace on disk. This may either be executed immediately (as in the case of an interactive workspace load), or in the future execution of a different user-defined workspace or function. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_shellcmd(cmd)**

File: codegen.c

Input: **char *cmd;**

Output: None

Function: **Cipe_shellcmd** generates an instruction to execute an arbitrary Unix shell command, such as to edit a file or to list the files in a directory. Such instructions may either be executed immediately (as in the case of an interactive *date* request) or in the future execution of a user-defined workspace or function. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_quit()**

File: codegen.c

Input: None

Output: None

Function: **Cipe_quit** generates an instruction to quit the current workspace, function, or terminal session. The instruction is executed either immediately (as in the case of an interactive QUIT command or CTRL D), or in the future as part of a user-defined workspace or function. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_symbols()**

File: codegen.c

Input: None

Output: None

Function: **Cipe_symbols** generates an instruction to list the currently defined symbols. This instruction may be executed either immediately (as in the case of an interactive SYMBOLS command), or in the future as part of a user-defined workspace or function. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_functions()**

File: codegen.c

Input: None

Output: None

Function: **Cipe_functions** generates an instruction to list the currently defined functions. This instruction may be executed either interactively (as in the case of a FUNCTIONS command) or in the future execution of a user-defined workspace of function. Note that execution of this instruction shows both builtin and user-defined functions, but only those functions available in the current context. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_traces()**

File: codegen.c

Input: None

Output: None

Function: **Cipe_traces** generates an instruction to show the currently set traces. This is executed either immediately (as in the case of a keyboard TRACES command), or in the future execution of a user-defined workspace or function. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_show(index)**

File: codegen.c

Input: **symbolnum index;**

Output: None

Function: **Cipe_show** generates an instruction to list the specified user-defined function. This is executed either immediately (as in the case of a keyboard SHOW command), or in the future execution of a user-defined workspace or function. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_define(name,codeseg,args,result,text,scope)**

File: codegen.c

Input:
char *name,*text;
struct code_seg_def *codeseg;
struct arglist_def *args;
unsigned int result,scope;

Output: None

Function: **Cipe_define** generates an instruction associating a name in the current context with a specified user-defined function. This is executed either immediately (as in the case of a keyboard DEFINE command), or in the future execution of a user-defined workspace or function which defines subordinate pieces of code. Note that functions defined using this instruction are defined only in the context of execution. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_print(exprs)**

File: codegen.c

Input: **struct exprlist_def *exprs;**

Output: None

Function: **Cipe_print** generates an instruction to display the value of a particular variable. This instruction is either executed immediately (as in the case of an interactive PRINT command), or in the future execution of a user-defined workspace of function. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_read_symbol(symbol,filename)**

File: codegen.c

Input: **symbolnum symbol;**
symbolnum filename;

Output: None

Function: **Cipe_read_symbol** generates an instruction to read a symbol value from a specified file. This instruction may either be executed immediately (as in the case of a keyboard READ command) or in the future execution of a user-defined workspace or function. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_write_symbol(symbol,filename)**

File: codegen.c

Input: **symbolnum symbol,filename;**

Output: None

Function: **Cipe_write_symbol** generates an instruction to write a symbol value to a specified file. This instruction may either be executed immediately (as in the case of a keyboard WRITE command) or in the future execution of a user-defined workspace or function. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_display(image,location)**

File: codegen.c

Input: **symbolnum image;**
struct location_def location;

Output: None

Function: **Cipe_display** generates an instruction to display a symbol at a specified location on a display device. This instruction may either be executed immediately (as in the case of a keyboard DISPLAY command) or in the future execution of a user-defined workspace or function. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_menu(on)**

File: codegen.c

Input: **int on;**

Output: None

Function: **Cipe_menu** generates an instruction to switch to or from using menus as the main user interface. This instruction may either be executed immediately (as in the case of a keyboard MENU command) or in the future execution of a user-defined workspace or function, but if executed in a workspace or function, will not take effect until execution completes. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_help()**

File: codegen.c

Input: None

Output: None

Function: **Cipe_help** generates an instruction to provide help. This is normally executed immediately as part of a keyboard HELP command, but might be useful in user-defined workspaces as well. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_save_scalar(scalar,type)**

File: codegen.c

Input: **unsigned int scalar;**
int type;

Output: None

Function: **Cipe_save_scalar** takes a scalar value and a value type, and stores the value in the current symbol table. A temporary variable (type *c*) is created and storage allocated. The function returns the symbol number (symbol table index) of the new variable, or **NO_SYMBOL** on error.

Suggested Modification:
None

Subroutine: **cipe_connect_constants(const1,const2)**

File: codegen.c

Input: **symbolnum const1,const2;**

Output: None

Function: **Cipe_connect_constants** takes two constants as specified by their symbol numbers (symbol table indices in the current context), and makes one multiple element constant of the two. Either of the original constants may have several values, but they are assumed to be of at most one dimension, i.e., scalars or vectors. The result is a vector whose size is the sum of the sizes of the original constants, and referenced by the symbol table location of the first component constant. The second constant is deleted. Data types are changed (e.g., from **int** to **float**) if necessary for compatibility. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_save_string(string)**

File: codegen.c

Input: **char *string;**

Output: None

Function: **Cipe_save_string** takes a string value and stores the value in the current symbol table. A temporary variable (type *c*) is created and storage allocated. The number of dimensions is set to 0, the number of bands and lines to 1, and the number of samples to the length of the string plus the terminating null byte. The data type is set to **STRING_TYPE**. The function returns the symbol number (symbol table index) of the new variable, or **NO_SYMBOL** on error.

Suggested Modification:
None

Subroutine: **cipe_save_keyword(keyword_num)**

File: codegen.c

Input: **unsigned int keyword_num;**

Output: None

Function: **Cipe_save_keyword** takes a keyword number and stores it as a value in the current symbol table. A temporary variable (type *c*) is created and storage allocated. Keywords are stored much as scalars, but are of type **KEYWORD_TYPE**. The function returns the symbol number (symbol table index) of the new variable, or **NO_SYMBOL** on error.

Suggested Modification:
None

Subroutine: **cipe_save_boolean(boolean_num)**

File: codegen.c

Input: **unsigned int boolean_num;**

Output: None

Function: **Cipe_save_boolean** takes a boolean value and stores it in the current symbol table. A temporary variable (type *c*) is created and storage allocated. Booleans are stored much as scalars, but are of type **BOOL_TYPE**. The function returns the symbol number (symbol table index) of the new variable, or **NO_SYMBOL** on error.

Suggested Modification:
None

Subroutine: **cipe_disassemble(address,buffer)**

File: codegen.c

Input: **struct instruction *address;**

Output: **char *buffer;**

Function: **Cipe_disassemble** writes a textual description of a compiled instruction to the specified buffer. The description includes the address of the instruction, the instruction mnemonic, and symbolic names of the instruction arguments and results. The function returns −1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_get_temp(type)**

File: codegen.c

Input: **char type;**

Output: None

Function: **Cipe_get_temp** creates a temporary symbol of the specified type and returns the symbol table index, or **NO_SYMBOL** on error. Temporary symbols come in a variety of types, depending on the application, and have names of the form *_xn*, where *x* is the type (e.g., *c* for constants encountered in compiling command lines or user-defined workspaces and functions) and *n* is a number. Numbers uniquely identify symbols, but symbols are referenced internally by the symbol table index.

Suggested Modification:
None

Subroutine: **cipe_istemp(symbol)**

File: codegen.c

Input: **symbolnum symbol;**

Output: None

Function: **Cipe_istemp** returns true (non-zero) if the specified symbol number (symbol table index) references a temporary symbol. This is equivalent to checking whether the first character of the symbol name is an underscore (_), since underscore is not a valid character for starting user-defined variable names.

Suggested Modification:
None

Subroutine: **cipe_save_instruction(iptr)**

File: codegen.c

Input: **struct instruction *iptr;**

Output: None

Function: **Cipe_save_instruction** takes a pointer to an instruction (as created by any of the instruction generation routines documented here) and stores it at the next location in the current code segment. The function first checks for sufficient storage, allocating more if necessary, then stores the instruction in the next available location, increments the number of instructions, and if requested by a previous *TURN ON CODEGEN TRACE* command, prints out a disassembled version of the instruction. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_clear_code()**

File: codegen.c

Input: None

Output: None

Function: **Cipe_clear_code** removes the code in the current code segment, resetting the next instruction pointer and the code length to their initial values.

Suggested Modification:
Allocated data structures used by the removed code should be properly freed.

Subroutine: **yylex()**

File: lexsrc

Input: None

Output: None

Function: A lexical analyzer, **yylex**, is generated by the Unix utility *lex* using the CIPE Cli file *lexsrc*. This function reads from the CIPE input buffers, tracing lexical analysis if requested using the *TURN ON LEX TRACE* command, separates the input into tokens, and then returns these tokens to the parser (*yyparse*) as needed. The function is called automatically by the parser, so no explicit calls by the programmer are necessary.

Suggested Modification:
None

Subroutine: **cipe_lex_init()**

File: lexsrc

Input: None

Output: None

Function: **Cipe_lex_init** performs initializations before lexical analysis. These include setting up and initializing buffer space. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_lex_push_stream(input_file,verbose)**

File: lexsrc

Input: **char *input_file;**
unsigned char verbose;

Output: None

Function: **Cipe_lex_push_stream** opens the specified input file and gets future command line input from there. A new stream description structure is created and the structure for the previous stream is pushed onto a stack. The verbose argument indicates whether or not input from the new stream should be echoed on the terminal. This is most useful when a command has been edited and the modified command needs to be redisplayed during execution. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_lex_prep_buffer()**

File: lexsrc

Input: None

Output: None

Function: **Cipe_lex_prep_buffer** prepares Cipe lexical analysis buffers to receive more input. It checks to make sure that a stream is still available to read from, and saves the last command in case the user wants to edit it. It then resets pointers and lexical analysis status variables and returns. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_lex_get_cmdline()**

File: lexsrc

Input: None

Output: None

Function: **Cipe_lex_get_cmdline** gets a command line from the current input stream and stores it in an internal lexical analysis buffer. If the current input stream is the standard input (i.e., input is being taken from the keyboard rather than from a workspace file or a temporary editor file), it prints the prompt; either the start-of-command prompt or the indented line-continuation prompt. If the user requests editing (using CTRL E), it calls for editing and the reading of the edited command as a new input stream. Otherwise, the keyboard input is simply stored. If input is coming from a file, the file is read and the input line saved; at end-of-file, the stream descriptor is popped off of the stack. Regardless of where input comes from, the function manages buffer space, increasing it if necessary, removes leading spaces and comments, and if in verbose mode, echos the input line to the user. The function returns the first character of the new input line.

Suggested Modification:
None

Subroutine: **cipe_lex_lookup_reserved_word(token,at_bol)**

File: lexsrc

Input: **char *token;**
int at_bol;

Output: None

Function: **Cipe_lex_lookup_reserved_word** takes a token and a flag indicating whether or not the token is the first on its line, and returns true (non-zero) if the token is a reserved word. The input token is first converted to lower case and if syntactically eligible, is checked against the list of known system attributes. System attributes containing an underscore may also be specified using a space and with component words abbreviated, requiring some local parsing to recognize correct specifications. If the token is not an attribute, it is checked against lists of other reserved words (e.g., *quit*) and booleans and keywords (e.g., *on*, *ivas*). The function returns –1 if not a reserved word, and the type of reserved word otherwise. The beginning-of-line flag is currently ignored.

Suggested Modification:
None

Subroutine: **cipe_lex_edit_input()**

File: lexsrc

Input: None

Output: None

Function: **Cipe_lex_edit_input** creates a temporary file and writes into it the current or last command. It then enables the user to edit the file using a standard system editor and causes the file to be set up as a new input stream. When execution of the file has finished, the file is deleted and this function returns. Note that in this case lexical analysis is actually recursive. If the command edited had invoked workspaces, several levels of lexical analysis would have been present at the same time. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **yyerror(s,bufpos)**

File: lexsrc

Input: **char *s,*bufpos;**

Output: None

Function: **Yyerror** is called automatically by the parser when an error message needs to be printed. If the current input stream is the standard input (i.e., input is coming from the keyboard), the offending line is displayed with a carat (^) underneath marking the error. If input is coming from a file, the filename and line number are displayed. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_lex_cleanup()**

File: lexsrc

Input: None

Output: None

Function: **Cipe_lex_cleanup** removes from the stack all pending workspace- and file-based input streams. Input is taken from the keyboard, subsequently. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
None

Subroutine: **cipe_ensure_buffer_space(nbytes)**

File: lexsrc

Input: **int nbytes;**

Output: None

Function: **Cipe_ensure_buffer_space** makes sure that the lexical analysis buffers are large enough to hold the specified number of additional characters. If the buffers are too small, more space is allocated. The initial buffer size is 1024 characters. More space may be needed in the case of long function definitions, which as single commands must fit into the buffers at once.

Suggested Modification:
None

Subroutine: **cipe_lt_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**

File: ops.c

Input: **char *lhs_data,*rhs_data;**
unsigned int datasize,datatype;

Output: **char *result_data;**

Function: **Cipe_lt_builtin** takes pointers to two scalars or vectors of the specified type and returns a boolean scalar or vector indicating, for each pair of values, their relationship. This is normally used by the executive when the size of the data items is not sufficient to justify downloading to a coprocessor. Input arguments are assumed to be of the same type. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
The function should accept types other than integer and float.

Subroutine: **cipe_lte_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**

File: ops.c

Input: **char *lhs_data,*rhs_data;**
unsigned datasize,datatype;

Output: **char *result_data;**

Function: **Cipe_lte_builtin** takes pointers to two scalars or vectors of the specified type and returns a boolean scalar or vector indicating, for each pair of values, their relationship. This is normally used by the executive when the size of the data items is not sufficient to justify downloading to a coprocessor. Input arguments are assumed to be of the same type. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
The function should accept types other than integer and float.

Subroutine: **cipe_eq_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**

File: ops.c

Input: **char *lhs_data,*rhs_data;**
unsigned datasize,datatype;

Output: **char *result_data;**

Function: **Cipe_eq_builtin** takes pointers to two scalars or vectors of the specified type and returns a boolean scalar or vector indicating, for each pair of values, their relationship. This is normally used by the executive when the size of the data items is not sufficient to justify downloading to a coprocessor. Input arguments are assumed to be of the same type. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
The function should accept types other than integer and float.

Subroutine: **cipe_ne_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**

File: ops.c

Input: **char *lhs_data,*rhs_data;**
unsigned datasize,datatype;

Output: **char *result_data;**

Function: **Cipe_ne_builtin** takes pointers to two scalars or vectors of the specified type and returns a boolean scalar or vector indicating, for each pair of values, their relationship. This is normally used by the executive when the size of the data items is not sufficient to justify downloading to a coprocessor. Input arguments are assumed to be of the same type. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
The function should accept types other than integer and float.

Subroutine: **cipe_gte_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**

File: ops.c

Input: **char *lhs_data,*rhs_data;**
unsigned datasize,datatype;

Output: **char *result_data;**

Function: **Cipe_gte_builtin** takes pointers to two scalars or vectors of the specified type and returns a boolean scalar or vector indicating, for each pair of values, their relationship. This is normally used by the executive when the size of the data items is not sufficient to justify downloading to a coprocessor. Input arguments are assumed to be of the same type. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
The function should accept types other than integer and float.

Subroutine: **cipe_gt_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**

File: ops.c

Input: **char *lhs_data,*rhs_data;**
unsigned datasize,datatype;

Output: **char *result_data;**

Function: **Cipe_gt_builtin** takes pointers to two scalars or vectors of the specified type and returns a boolean scalar or vector indicating, for each pair of values, their relationship. This is normally used by the executive when the size of the data items is not sufficient to justify downloading to a coprocessor. Input arguments are assumed to be of the same type. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
The function should accept types other than integer and float.

Subroutine: **cipe_and_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**

File: ops.c

Input: **char *lhs_data,*rhs_data;**
unsigned datasize,datatype;

Output: **char *result_data;**

Function: **Cipe_and_builtin** takes pointers to two scalars or vectors of the specified type and returns a boolean scalar or vector indicating, for each pair of values, whether or not they are both non-zero. This is normally used by the executive when the size of the data items is not sufficient to justify downloading to a coprocessor. Input arguments are assumed to be of the same type. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
The function should accept types other than 4-byte integer.

Subroutine: **cipe_or_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**

File: ops.c

Input: **char *lhs_data,*rhs_data;**
unsigned datasize,datatype;

Output: **char *result_data;**

Function: **Cipe_or_builtin** takes pointers to two scalars or vectors of the specified type and returns a boolean scalar or vector indicating, for each pair of values, whether or not either of the values is non-zero. This is normally used by the executive when the size of the data items is not sufficient to justify downloading to a coprocessor. Input arguments are assumed to be of the same type. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
The function should accept types other than 4-byte integer.

Subroutine: **cipe_bitand_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**

File: ops.c

Input: **char *lhs_data,*rhs_data;**
unsigned datasize,datatype;

Output: **char *result_data;**

Function: **Cipe_bitand_builtin** takes pointers to two scalars or vectors of the specified type and returns a boolean scalar or vector indicating, for each pair of values, the value of their bit-wise AND. This is normally used by the executive when the size of the data items is not sufficient to justify downloading to a coprocessor. Input arguments are assumed to be of the same type. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
The function should accept types other than 4-byte integer.

Subroutine: **cipe_bitor_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**

File: ops.c

Input: **char *lhs_data,*rhs_data;**
unsigned datasize,datatype;

Output: **char *result_data;**

Function: **Cipe_bitor_builtin** takes pointers to two scalars or vectors of the specified type and returns a boolean scalar or vector indicating, for each pair of values, the value of their bit-wise OR. This is normally used by the executive when the size of the data items is not sufficient to justify downloading to a coprocessor. Input arguments are assumed to be of the same type. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
The function should accept types other than 4-byte integer.

Subroutine: **cipe_bitxor_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**

File: ops.c

Input: **char *lhs_data,*rhs_data;**
unsigned datasize,datatype;

Output: **char *result_data;**

Function: **Cipe_bitxor_builtin** takes pointers to two scalars or vectors of the specified type and returns a boolean scalar or vector indicating, for each pair of values, the value of their bit-wise XOR. This is normally used by the executive when the size of the data items is not sufficient to justify downloading to a coprocessor. Input arguments are assumed to be of the same type. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
The function should accept types other than 4-byte integer.

Subroutine: **cipe_plus_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**

File: ops.c

Input: **char *lhs_data,*rhs_data;**
unsigned datasize,datatype;

Output: **char *result_data;**

Function: **Cipe_plus_builtin** takes pointers to two scalars or vectors of the specified type and returns a boolean scalar or vector indicating, for each pair of values, their sum. This is normally used by the executive when the size of the data items is not sufficient to justify downloading to a coprocessor. Input arguments are assumed to be of the same type. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
The function should accept types other than integer and float.

Subroutine: **cipe_minus_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**

File: ops.c

Input: **char *lhs_data,*rhs_data;**
unsigned datasize,datatype;

Output: **char *result_data;**

Function: **Cipe_minus_builtin** takes pointers to two scalars or vectors of the specified type and returns a boolean scalar or vector indicating, for each pair of values, their difference. This is normally used by the executive when the size of the data items is not sufficient to justify downloading to a coprocessor. Input arguments are assumed to be of the same type. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
The function should accept types other than integer and float.

Subroutine: **cipe_mult_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**

File: ops.c

Input: **char *lhs_data,*rhs_data;**
unsigned datasize,datatype;

Output: **char *result_data;**

Function: **Cipe_mult_builtin** takes pointers to two scalars or vectors of the specified type and returns a boolean scalar or vector indicating, for each pair of values, their product. This is normally used by the executive when the size of the data items is not sufficient to justify downloading to a coprocessor. Input arguments are assumed to be of the same type. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
The function should accept types other than integer and float.

Subroutine: **cipe_div_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**

File: ops.c

Input: **char *lhs_data,*rhs_data;**
unsigned datasize,datatype;

Output: **char *result_data;**

Function: **Cipe_div_builtin** takes pointers to two scalars or vectors of the specified type and returns a boolean scalar or vector indicating, for each pair of values, their quotient. This is normally used by the executive when the size of the data items is not sufficient to justify downloading to a coprocessor. Input arguments are assumed to be of the same type. The function returns –1 on error, and 0 otherwise.

Suggested Modification:
The function should accept types other than integer and float.

Subroutine: **yyparse()**

File: parser.y

Input: None

Output: None

Function: **Yyparse** is a parser generated by the Unix *yacc* utility using the Cipe Cli source file *parser.y*. It performs all command line parsing (calling *yylex* for tokens as necessary) and the corresponding error detection, calling instruction generation routines to create pseudo code in the current code segment. These instructions are then executed either immediately or in the future invocation of a user-defined workspace or function. The function provides for several type of tracing, as specified by the user.

Suggested Modification:
None

Appendix C – HOST RESIDENT EXECUTIVE SUBROUTINES

C.1 Symbol Manager

Subroutine: **cipe_creat_symbol(sname,sptr,sindex)**

File: symbol.c

Input: **char *sname (symbol name);**

Output: **struct cipe_symbol *sptr;**
int sindex (symbol array inpdex);

Function: Creates a symbol entry in the symbol table under the symbol name **sname** and allocates a memory space for a symbol structure. Initializes each field of the symbol structure with default values and returns the pointer to the symbol structure (**sptr**) and the index (**sindex**) of the symbol table entry.

Suggested Modification:
Simplify calling sequence to sindex = cipe_creat_symbol(sname).

Subroutine: **struct cipe_symbol *cipe_get_symbol(sname)**

File: symbol.c

Input: **char *sname;**

Output: **struct cipe_symbol *sptr;**

Function: Returns a symbol structure pointer of a given symbol name.

Suggested Modification:
Simplify calling sequence to sindex = cipe_creat_symbol(sname).

Subroutine: **cipe_delete_symbol(sindex)**

File: symbol.c

Input: **int sindex;**

Output: None

Function: Deletes a symbol entry from the symbol table, deallocates the symbol structure, and reorganizes the symbol table.

Suggested Modification:
None

Subroutine: **struct cipe_symbol *cipe_symbol_index_to_ptr(sindex)**

File: symbol.c

Input: **int sindex;**

Output: None

Function: Returns a symbol structure pointer to a symbol in the symbol table indexed by **sindex**.

Suggested Modification:
None

Subroutine: **char *cipe_symbol_index_to_name(sindex)**
File: symbol.c
Input: **int sindex;**
Output: None
Function: Returns a symbol name of a symbol in the symbol table entry indexed by **sindex**.
Suggested Modification:
None

Subroutine: **cipe_dump_symbol_table(verbose)**
File: symbol.c
Input: **int verbose;**
Output: None
Function: Lists out symbol information.
Suggested Modification:
None

Subroutine: **cipe_func_delete_symbol(sname)**
Input: **char *sname;**
Output: None
Function: Deletes the symbol entry from the symbol table in the host and the nodes.
Suggested Modification:
Change sname into sindex.

Subroutine: **cipe_clear_temp_symbols()**
File: symbol.c
Input: None
Output: None
Function: Deletes temporary symbols from the symbol table.
Suggested Modification:
None

C.2 Program Manager

Subroutine: **cipe_func_application(function_name)**
File: doload.c
Input: **char *function_name;**
Output: None
Function: Loads an application function and executes the function.
Suggested Modification:
None

Subroutine: **cipe_load_module(function_file)**
File: doload.c
Input: **char *function_file;**
Output: None
Function: Loads the executable application program file **function_file** into the predefined program area.
Suggested Modification:
None

Subroutine: **cipe_execute_module()**
File: doload.c
Input: None
Output: None
Function: Jumps to the program area and starts execution.
Suggested Modification:
None

C.3 File I/O

Subroutine: **cipe_func_read_from_file(sname,header)**

File: dofile.c

Input: **char *sname;**
strcut image_hdr *header;

Output: None

Function: Reads the file whose name is specified in the filename field of the symbol **sname**. Any subset of the file can be specified in the area description fields of the symbol (sl,ss,sb,nl,ns,nb). The data is read into the malloced symbol data area.

Suggested Modification:
Instead of using a symbol name, a symbol table entry index should be used.

Subroutine: **cipe_func_write_to_file(filename,sname)**

File: dofile.c

Input: **char *filename;**
char *sname;

Output: None

Function: Writes the data of the symbol **sname** to the file **filename.**

Suggested Modification:
Instead of using a symbol name, a symbol table entry index should be used.

Subroutine: **struct image_hdr *cipe_get_image_header(filename)**

File: dofile.c

Input: **char *filename;**

Output: None

Function Opens a header file (filename.hdr), parses the header information, and returns a pointer to the image_header structure.

Suggested Modification:
None

Subroutine: **cipe_put_image_header(filename,header)**

File: **dofile.c**

Input: **char *filename;**
struct image_hdr header;

Output: None

Function: Creates a header file (**filename.hdr**) and writes the header buffer into the header file.

Suggested Modification:
None

Subroutine: **cipe_open(filename,option)**
File: dofile.c
Input: **char *filename;**
int option;
Output:
Function: Same as Unix open.
Suggested Modification:
None

Subroutine: **cipe_creat(filename,header)**
File: dofile.c
Input: **char *filename;**
struct image_hdr *header;
Output: None
Function: Executes **cipe_put_image_header(filename,header)** and the Unix command **creat(filename,0644)** to create a file header and file.
Suggested Modification:
None

Subroutine: **cipe_read(filename,buffer,nbytes)**
File: dofile.c
Input: **char *filename;**
int nbytes;
Output: **unsigned char *buffer**
Function: Same as Unix read.
Suggested Modification:
None

Subroutine: **cipe_write(filename,buffer,nbytes)**
File: dofile.c
Input: **char *filename;**
int nbytes;
Output: **unsigned char *buffer**
Function: Same as Unix write.
Suggested Modification:
None

C.4 Display

Subroutine: **cipe_func_draw(plane,sname,loc)**

File: dodisp.c

Input:
```
int plane;
char *sname;
int loc[2];
```

Output: None

Function: Draws the data of the symbol **sname** on the image plane **plane** starting at the location described by **loc[0]** (line) and **loc[1]** (sample). When the data type is not character type, the data type is converted prior to display.

Suggested Modification:
Instead of symbol name, symbol table entry index may be used.

Subroutine: **cipe_func_mssdraw(sname,band,loc)**

File: dodisp.c

Input:
```
int plane;
char *sname;
```

Output: None

Function: Draws a selected band (**band**) of the multispectral data (three dimensional) of the symbol **sname** starting at the location described by **loc[0]** (line) and **loc[1]** (sample). When the data type is not character type, the data type is converted prior to display.

Suggested Modification:
Instead of symbol name symbol table entry index may be used

Subroutine: **cipe_func_mssplot(sname,xrange,yrange)**

File: doplot.c

Input:
```
char *sname;
float xrange[2];
int yrange[2];
```

Output: None

Function: This routine is an interactive plotting routine where a user can continue plotting a specified subarea of the specified dataset (**sname**). Each spectrum will be plotted with different color using x axis as wavelength range and y axis as intensity value range.

Suggested Modification:
None

Subroutine: **cipe_func_plot3d(input,viewelev,viewazi,plotoffsets)**

File: doplot3d.c

Input:
```
char *input;
float viewelev;
float viewazi;
float plotoffsets[2];
```

Output: None

Function: This routine plots image data using the intensity as z axis. For specified elevation and azimuth of a viewer (**viewelev, viewazi**), the projected input data are plotted with the **plotoffsets** as an origin of coordinates(0,0,0).

Suggested Modification:
None

C.5 Hypercube Interface

Subroutine: **cipe_cube_builtin(function_name,lhs,rhs,result)**

File: docube.c

Input: **int function_name;**
char *lhs;
char *rhs;

Output: **char *result**

Function: Performs an operation(+,-,*,&,^,|,&&,||,...) specified by the **function_name** on two symbols (lhs - left-hand side symbol name, rhs- right-hand side symbol name) and stores the result in the output symbol **result.**

Suggested Modification:
None

Subroutine: **cipe_cube_execute_module(module_name)**

File: docube.c

Input: **char *module_name;**

Output: None

Function: Loads hypercube resident application program module into the hypercube and executes the program.

Suggested Modification:
None

Subroutine: **cipe_cube_compose_loadmap(sptr)**

File: docube.c

Input:

Output: None

Function: Composes a loadmap for a symbol pointed by the **sptr** according to the specified load-type and its data characteristics.

Suggested Modification:
None

Subroutine: **cipe_cube_read_loadmap(sptr)**

File: docube.c

Input: **struct cipe_symbol *sptr;**

Output: None

Function: Reads a loadmap of a symbol pointed by **sptr** from the cube.

Suggested Modification:
None

Subroutine: **cipe_cube_write_loadmap(sptr)**
File: docube.c
Input: **struct cipe_symbol *sptr;**
Output: None
Function: Writes a loadmap of a symbol pointed by **sptr** to the cube.
Suggested Modification:
None

Subroutine: **cipe_cube_read_data(sptr)**
File: docube.c
Input: **struct cipe_symbol *sptr;**
Output: None
Function: Reads data of a symbol pointed by **sptr** from the cube.
Suggested Modification:
None

Subroutine: **cipe_cube_write_data(sptr,loadtype)**
File: docube.c
Input: **struct cipe_symbol *sptr;**
int loadtype;
Function: Writes data of a symbol pointed by sptr to the cube according to the loadtype.
Suggested Modification:
None

Appendix D – HYPERCUBE RESIDENT EXECUTIVE SUBROUTINES

Subroutine: **elt_init( )**
File: elt_mon.c
Input: none
Output: none
Function: Initializes some global hypercube parameters, sets up the symbol table, and initializes the type size array.
Suggested Modification:
None

Subroutine: **elt_panic( )**
File: elt_mon.c
Input: **char *str**
Output: none
Function: Terminates node monitor, prints a corresponding message, and turns on the light emitting diodes (leds) on the hypercube.
Suggested Modification:
None

Subroutine: **getstring(str)**
File: elt_module.c
Input: none
Output: **char *str**
Function: Hypercube gets a string from the control processor and stores it in **str**.
Suggested Modification:
None

Subroutine: **getint(iptr)**
File: elt_module.c
Input: none
Output: **int *iptr**
Function: Hypercube gets an integer from the control processor and stores it in **iptr**.
Suggested Modification:
None

Subroutine: **elt_load_module( )**
File: elt_module.c
Input: none
Output: **int td_size**
int bss_size
Function: Hypercube receives a module for execution.
Suggested Modification:
None

Subroutine: **elt_execute_module( )**
File: elt_module.c
Input: none
Output: none
Function: Begins execution of the current module.
Suggested Modification:
None

Subroutine: **elt_init_symtab( )**
File: elt_symbol.c
Input: none
Output: none
Function: Initializes the symbol table.
Suggested Modification:
None

Subroutine: **SYMBOL *elt_get_symbol(name,global)**
File: elt_symbol.c
Input: **char *name**
int global

Output: none
Function: Goes and gets the symbol with name **name.**
Suggested Modification:
None

Subroutine: **SYMBOL *elt_create_symbol(name,global,size,datatype)**
File: elt_symbol.c
Input: **char *name**
int global
int size
int datatype

Output: none
Function: Creates a symbol with name **name** and returns a pointer to it.
Suggested Modification:
None

Subroutine: **SYMBOL *elt_delete_symbol(sym)**
File: elt_symbol.c
Input: **SYMBOL *sym**
Output: none
Function: Deletes a symbol from the symbol table.
Suggested Modification:
None

Subroutine: **elt_lock_global(sym)**
File: elt_symbol.c
Input: **SYMBOL *sym**
Output: none
Function: Locks a global symbol to prevent its data from being moved during memory compaction.
Suggested Modification:
None

Subroutine: **elt_unlock_global(sym)**
File: elt_symbol.c
Input: **SYMBOL *sym**
Output: none
Function: Unlocks a global symbol to allow its data to be moved during memory compaction.
Suggested Modification:
None

Subroutine: **elt_builtin( )**
File: elt_bltin.c
Input: none
Output: none
Function: Runs in the hypercube to set up for and call the builtin function the user requested.
Suggested Modification:
None

Subroutine: **elt_lt_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**
File: elt_bltin.c
Input: **unsigned char *lhs_data, *rhs_data**
int datasize, datatype

Output: **unsigned char *result_data**
Function: Less than function.
Suggested Modification:
None

Subroutine: **elt_lte_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**
File: elt_bltin.c
Input: **unsigned char *lhs_data, *rhs_data**
int datasize, datatype

Output: **unsigned char *result_data**
Function: Less than or equal to function.
Suggested Modification:
None

Subroutine: **elt_eq_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**
File: elt_bltin.c
Input: **unsigned char *lhs_data, *rhs_data**
int datasize, datatype

Output: **unsigned char *result_data**
Function: Equal function.
Suggested Modification:
None

Subroutine: **elt_ne_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**
File: elt_bltin.c
Input: **unsigned char *lhs_data, *rhs_data**
int datasize, datatype

Output: **unsigned char *result_data**
Function: Not equal function.
Suggested Modification:
None

Subroutine: **elt_gte_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**
File: elt_bltin.c
Input: **unsigned char *lhs_data, *rhs_data**
int datasize, datatype

Output: **unsigned char *result_data**
Function: Greater than or equal to function.
Suggested Modification:
None

Subroutine: **elt_gt_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**
File: elt_bltin.c
Input: **unsigned char *lhs_data, *rhs_data**
int datasize, datatype

Output: **unsigned char *result_data**
Function: Greater than function.
Suggested Modification:
None

Subroutine: **elt_and_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**
File: elt_bltin.c
Input: **unsigned char *lhs_data, *rhs_data**
int datasize, datatype

Output: **unsigned char *result_data**
Function: AND function.
Suggested Modification:
None

Subroutine: **elt_or_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**
File: elt_bltin.c
Input: **unsigned char *lhs_data, *rhs_data**
int datasize, datatype

Output: unsigned char *result_data
Function: OR function.
Suggested Modification:
None

Subroutine: **elt_bitand_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**
File: elt_bltin.c
Input: **unsigned char *lhs_data, *rhs_data**
int datasize, datatype

Output: **unsigned char *result_data**
Function: Bitwise AND function.
Suggested Modification:
None

Subroutine: **elt_bitor_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**
File: elt_bltin.c
Input: **unsigned char *lhs_data, *rhs_data**
int datasize, datatype

Output: **unsigned char *result_data**
Function: Bitwise OR function.
Suggested Modification:
None

Subroutine: **elt_bitxor_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**
File: elt_bltin.c
Input: **unsigned char *lhs_data, *rhs_data**
int datasize, datatype

Output: **unsigned char *result_data**
Function: Bitwise XOR function.
Suggested Modification:
None

Subroutine: **elt_plus_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**
File: elt_bltin.c
Input: **unsigned char *lhs_data, *rhs_data**
int datasize, datatype

Output: **unsigned char *result_data**
Function: Addition function.
Suggested Modification:
None

Subroutine: **elt_minus_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**
File: elt_bltin.c
Input: **unsigned char *lhs_data, *rhs_data**
int datasize, datatype

Output: **unsigned char *result_data**
Function: Subtraction function.
Suggested Modification:
None

Subroutine: **elt_mult_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**
File: elt_bltin.c
Input: **unsigned char *lhs_data, *rhs_data**
int datasize, datatype

Output: **unsigned char *result_data**
Function: Multiplication function.
Suggested Modification:
None

Subroutine: **elt_div_builtin(lhs_data,rhs_data,result_data,datasize,datatype)**
File: elt_bltin.c
Input: **unsigned char *lhs_data, *rhs_data**
int datasize, datatype

Output: **unsigned char *result_data**
Function: Division function.
Suggested Modification:
None

Appendix E – HYPERCUBE DATA MANAGER SUBROUTINES

Subroutine: **elt_read_data(symbol)**

File: elt_data.c

Input: **struct elt_symbol *symbol**

Need the desired data decomposition and the load map which are stored in the structure.

Output: **struct elt_symbol *symbol**

Function: This subroutine runs in the nodes of the hypercube to read data from the control processor and place it in the nodes according to the decomposition type and the load map stored in the symbol. It communicates with the corresponding control processor routine, **cube_write( )**.

Suggested Modification:
None

Subroutine: **elt_write_data(symbol)**

File: elt_data.c

Input: **struct elt_symbol *symbol**

Reads the old load map from the control processor.

Output: Image data is sent back to the control processor.

Function: This routine, which runs in the hypercube, sends data from the nodes to the control processor. The decomposition type of the current data is taken from the old load map. The desired manner of sending back the data is taken from the decomposition type and the load map stored in **symbol**. This subroutine communicates with the corresponding control processor subroutine, **cube_read( )**.

Suggested Modification:
Further test returning the data using a CUSTOM decomposition.

Subroutine: **elt_redist_data(symbol)**

File: elt_data.c

Input: **struct elt_symbol *symbol**

Reads old load map from control processor.

Output: **struct elt_symbol *symbol**

Function: This routine runs in the hypercube to move data between decompositions within the cube. Only certain transitions have been implemented and some may never be possible. Load type for current data is taken from the old load map, and data is redistributed according to the load type and load map stored in **symbol**. This subroutine communicates with the corresponding control processor subroutine, **cube_redist( )**.

Suggested Modification:
none

Subroutine: **mv_horiz_grid_vert(symbol, oldloadmap, exchchan)**

File: elt_data.c

Input: **struct elt_loadmap *oldloadmap**
int exchchan

Output: **struct elt_symbol *symbol**
struct elt_loadmap *oldloadmap

Function: This subroutine runs in the hypercube to move data from a horizontal decomposition to a grid decomposition, or from a grid decomposition to a vertical decomposition. It reads the current data decomposition type and load map from **oldloadmap**, exchanges appropriate data along the channel specified in **exchchan**, and alters **oldloadmap** to reflect the data that is now stored in **symbol**.

Suggested Modification:
none

Subroutine: **mv_vert_grid_horiz(symbol,oldloadmap,exchchan)**

File: elt_data.c

Input: **struct elt_loadmap *oldloadmap**
int exchchan

Output: **struct elt_symbol *symbol**
struct elt_loadmap *oldloadmap

Function: Moves data from a vertical decomposition to a grid decomposition, or from a grid decomposition to a horizontal decomposition. Reads current data decomposition type and load map from **oldloadmap**, exchanges appropriate data along the channel specified in **exchchan**, and alters **oldloadmap** to reflect the data now stored in **symbol**.

Suggested Modification:
none

Subroutine: **cube_read(symbol,oldloadmap)**

File: cp_data.c

Input: **struct cipe_symbol *symbol**
struct loadmap *oldloadmap

Output: Image data is returned from the nodes of the hypercube to the control processor.

Function: Runs in the control processor to read image data from the hypercube back to the control processor. The hypercube uses **oldloadmap** to know how the data is stored in symbol, and uses the load map and decomposition type information stored in **symbol** to know how the user desires to read the data back.

Suggested Modification:
none

Subroutine: **cube_write(symbol,oldloadmap)**

File: cp_data.c

Input: **struct cipe_symbol *symbol**
struct loadmap *oldloadmap

Output: **struct cipe_symbol *symbol**

Function: Data is read from the control processor and stored in the nodes of the hypercube according to the load type and load map stored in **symbol**. **Oldloadmap** contains information for the entire image.

Suggested Modification:
none

Subroutine: **cube_redist(symbol,oldloadmap)**

File: cp_data.c

Input: **struct cipe_symbol *symbol**
struct loadmap *oldloadmap

Output: **struct cipe_symbol *symbol**

Function: Within the cube data is moved from one decomposition to another. The control processor sends **oldloadmap** to the cube to inform it how the data is currently stored. The desired data decomposition and load map are stored in **symbol**.

Suggested Modification:
none

Appendix F – IMAGE PROCESSING FUNCTION LIBRARY

Built-in Functions

Arithmetic Operations	: +, -, *, /
Logical Operations	: ==, !=, >, <, &&, ||
Bit manipulations	: &, |, ^,

Display Functions

Allocation	: B&W, Color, Pseudo
Stretch	: Linear, Table
Draw	: B&W, Color
MssDraw (multispectral data)	
Zoom	
Histogram	
Cursor	: Read, Write
Erase	: Image, Graphics

Plot Functions

Mssplot
3dplot

Image Enhancement

Spatial convolution filter
Histogram stretch
Frequency domain windowing (under development)

Geometric correction

Tie point
Map projection
Size

Data Classification

Spectral classification
Textural classification (under development)

Appendix G – YET ANOTHER MENU MANAGER (YAMM) PROGRAMMER'S GUIDE

1. INTRODUCTION

One of the most time-consuming yet necessary tasks of writing any piece of interactive software is the development of a user-interface. Frequently, the development of even a very simple interface takes up valuable time that might be much better devoted to the design and implementation of the underlying task. Yamm (*Y*et *a*nother *m*enu-*m*anager) is an application-independent menuing package, designed to remove much of the difficulty and save much of the time inherent in the implementation of front-ends for large software packages.

This paper gives an overview of Yamm's structure and design philosophy. Section 2 describes the menu configuration specification, a potentially user-specific description of an application's menu structure(s) and terminal interface. Section 3 describes the Yamm routines usable by an application. The steps involved in implementing a program using Yamm are covered in Section 4, and an example tieing everything together follows in Section 5.

2. MENU CONFIGURATION

Applications running under the menu package consist of two parts: a description of the menu configuration (sometimes including terminal characteristics), which may be user-specific, and the body of application code, which includes a call to start up the menu package as well as calls to menu package interface routines.

The menu configuration is used at runtime to define the menu structure and any non-standard terminal characteristics; it may be read in from a programmer-specified file or included in the source. The example program in Section 5 uses both methods: if the menu configuration file (*menuconfig*) exists, the program uses it, otherwise it relies on the coded version.

Menu configuration specifications are composed of menu and terminal definitions.

Menu definitions define specific menus within the menu tree. Each definition contains the word *MENU* and a menu name, a series of prompt-name pairs, and an *END*. The name in each pair may be either a reference to an application function or the name of another menu defined within the menu configuration.

Terminal definitions are optional and allow the user to specify non-standard keyboard mappings and terminal capabilities. Each definition contains the word *TTY* and a series of terminal types (e.g., *vt100 sun*) followed by key and capability definitions. Key definitions specify a generic key-name (e.g., *up_arrow*, *help*), what the user calls the key to be used (e.g., *go up*, *CTRL H*), and the actual ASCII sequence generated by the key. For example:

help:CTRL H:^H

The redefinable keys are:

up_arrow	help	enddataentry	func_0	func_4
down_arrow	root	endselect	func_1	func_5
right_arrow	previous	refresh	func_2	func_6
left_arrow	exit	toggle	func_3	func_7

Terminal capability definitions give the capability name and then the corresponding key sequence, as in:

clear_screen::\033H\033J # ESC H ESC J – for VT52

The redefinable terminal capabilities are:

move	set_inv_video	reset_inv_video	clear_screen

set_underline	reset_underline	down_line	clear_eol
scroll_region	configure	reconfigure	set_gcs
reset_gcs	upper_left_cor	horiz_bar	upper_right_cor
vert_bar	lower_right_cor	lower_left_cor	

Character sequences may be formed using the usual \octal-value (e.g., *\033* for escape) and inserting the actual control or escape characters into the definition. The carat (^) may also be used to form easily-readable control characters. If system-special characters such as CTRL-C or CTRL-S are used, their normal functions are disabled inside the program.

Null padding may be specified for terminal capability definitions by preceding the character sequence with the number of following nulls. For example, if the definition for *clear_screen* above required a following padding of 10 nulls, the sequence could be changed to *10\033H\033J*. Neither type of terminal definition may be used in the menuconfig of a dynamically-created (nested) menu structure.

Note that the formats for the *move* and *scroll_region* sequences are different from those used by termcap. Specifically, the values must come in the reverse of the default termcap order, and start at 1 rather than 0. This simplifies integration of other terminals but may be confusing when compared with termcap entries.

Comments and white space may be used freely within the menu configuration commands; see the example for more information.

3. APPLICATION ROUTINES

Here are the functions callable from application programs. Most of these return –1 on error and 0 otherwise; see the example for information on actual usage. The structures and definitions used here are defined in menuapp.h.

```
menu_start(config_name,config,func_table,localinit,localreinit,logfile)
char *config_name;
char *config[];
struct func_name_pair func_table[];
int (*localinit)(),(*localreinit)();
FILE *logfile;
```

invokes the menu interface. It must be called before any of the other functions listed here. *Config_name* gives the name of the menu configuration file; this may be *NULL* if the configuration is hardcoded and no user override is to be permitted. *Config* gives a hardcoded default configuration; this may be *NULL* if there will always be a configuration file. *Func_table* is a list of function name-code pairings; this is illustrated below. *Localinit* and *localreinit* point to functions invoked automatically when the specified menu tree is entered and left, respectively. Either or both of these may be *NULL*, but if they are used, should return –1 on error and 0 otherwise. If session logging is desired, *logfile* should contain a file pointer to an open file. Otherwise it should be *NULL*. If logging is used, closing the file is left to the application.

```
getpar(params,num_params,error_code,help_code)
struct param *params;
int num_params;
int (*error_code)(),(*help_code)();
```

requests parameter values from the user using a data-entry screen. *Params* points to a series of

parameter definition structures, *num_params* contains the number of parameters defined, *error_code* points to a function able to detect errors or is *NOECHK* if none, and *help_code* points to a function which prints parameter information for the user on request or is *NOHELP* if none.

Parameter definition is done using *def( )*, which is described below.

The programmer-supplied error code receives three parameters: a pointer to the parameter definition structures, the number of the parameter to check, and the value supplied, an *int-double-char* * union. (Booleans are treated as integers for purposes of error checking.) Error checking functions typically use *unspecified( )*, *new_value( )*, and *unspecify( )*, described below, to detect and correct errors. The function must return -1 if the parameter value is unacceptable, and 0 otherwise.

Help code receives two parameters: a pointer to the parameter definition structures, and the number of the parameter in question. One additional special case is also provided for. On entry into *getpar( )*, Yamm calls the help routine specifying parameter number -1 to enable output of any initial help or welcome message. This is illustrated in the example.

```
def(params,param_num,prompt,type,value_ptr,...)
struct param *params;
int param_num;
char *prompt;
int type;
unsigned char *value_ptr;
```

defines parameters on the data-entry screen. The first five parameters are required, in the order shown; subsequent parameters may come in any order and are optional, with appropriate defaults.

params is the name of a block of storage reserved for parameters, as in *struct param params[5];* to reserve space for five parameters,

param_num is the number of the parameter being defined, starting with 0,

prompt is a pointer to a short textual description of the value which will be used as a prompt string,

type is the parameter type, *INT*, *FLOAT*, *DOUBLE*, *STRING*, or *BOOL* (YES/NO), and

value_ptr is a memory location to receive the specified value(s).

In addition, a variety of parameters may be defined by specifying a keyword followed by a comma and the relevant value(s):

INCR precedes an increment in bytes from *value_ptr* to use when multiple values are specified – the default is parameter-type dependent: 4 for INT and FLOAT, 8 for DOUBLE, 0 for STRING, and 4 for BOOL,

WID precedes the field width in columns needed to specify the value – the default is parameter-type dependent: 6 for INT, 8 for FLOAT and DOUBLE, 20 for STRING, and 3 for BOOL,

COUNT precedes a pointer to an integer variable which will receive the number of values specified – the default is to not supply a count,

LINE precedes the line location (where 0 is top of window) of the prompt – the default is 0,

COL precedes the column location of the prompt (where 0 is the left edge) – the default is 0,

DUP precedes the number of values expected, or the negative of the maximum number of values if the number is variable – the default is 1,

DEFAULT indicates that the area referenced by the *value_ptr* parameter contains default values which should be displayed to the user – the default is "no default,"

GROUP precedes the group number (greater than or equal to zero) of the parameter; if the user specifies a value for any parameter within the group, all parameters must be specified and for tables, match in number of values – the default is "no group,"

REQ indicates that a value for the parameter is required – the default is "not required,"

ENUM precedes two values, a number, and a pointer to that many values, from among which the user must choose in valuing the parameter; when the values are STRINGs (for STRING-type parameters), the pointer is to a list of pointers to these values, i.e., the list is defined: *static char *legal_values[] = {"value1","value2",...};*

ENTRYLINE precedes the line number (starting with 0) of the first data-entry blank for this parameter – the default is to use the line number of the prompt, except when multiple data items are grouped together in a table format, in which case the default is the number of the line below the prompt,

ENTRYCOL precedes the column number (starting with 0) of the first data-entry blank for this parameter – the default is that column to the right of the prompt, except when multiple data items are grouped together in a table format, in which case the default is the column number such that the prompt is centered over the column, and

END functions as an end-of-definition keyword and is required.

See the example for numerous parameter definitions.

unspecified(param)
int param;

returns true if no value has been specified by the user for parameter *param*. This is most useful in the application-supplied error-checking routine, where it can be used to make sure that a specified value does not conflict with other related parameter values. This is typically used with *new_value( )* and *unspecify( )* below.

```
new_value(params,entry)
struct param *params;
int entry;
```

notifies the menu package that the value of the parameter numbered by *entry* has been changed, usually because of a change in the value of a related parameter.

```
unspecify(param)
int param;
```

removes the value of the specified parameter. This is usually only useful inside error-checking routines, when a newly specified value is incompatible with the value of another parameter and the latter must be removed.

```
create_new_subtree(config_name,config,func_table,localinit,localreinit)
char *config_name;
char *config[];
struct func_name_pair func_table[];
int (*localinit)(),(*localreinit)();
```

dynamically creates a supplemental menu tree. The arguments are similar to the arguments for *menu_start( )* above. The code pointed to by *localinit* will be executed immediately, and if executed without error (i.e., code returns 0) then the user will be placed at the root of the new menu tree. The function pointed to by *localreinit* is run on exit. Recursion is permissible with care, but the use of *static* variables should be avoided.

4. TO IMPLEMENT A PROGRAM USING YAMM

The following steps are sufficient to implement most programs using Yamm:

1. Determine where the menu configuration file, if any, will be kept, and if desired, create a default configuration. Next, declare the functions which will be called through the menus and assign to each menu-callable function a name. Write any application-specific initialization and reinitialization code. If logging is desired, *fopen* a file to contain the log. Finally, write the call to *menu_start( )*, using the data structures above as parameters.

   Example:

```
char *config_name = "./menuconfig";
char *config[] = NULL;

int function1(),function2(),function3();

struct func_name_pair funcs[]={
    {function1,"func1"},
    {function2,"func2"},
    {function3,"func3"},
    {0,"/keep last"},
};

my_init() { /* application init code here */ }
my_reinit() { /* reinit code here */ }
```

```
main() { menu_start(config_name,config,funcs,my_init,my_reinit,NULL); }
```

2. Create a menu configuration file. An example for the above function table might be:

```
MENU mainmenu
DoFunc1/func1          # DoFunc1 is the prompt; if chosen, calls func1.
OtherFuncs/submenu
END

MENU submenu
DoFunc2/func2          # func2 and func3 must be associated with a
DoFunc3/func3          # function using a function table as above.
END
```

3. In each application routine called by the menus, define parameters using *def( )*, and write *chk_* and *help_* routines. (You may want to leave writing these routines until the application runs correctly. Use *NOECHK* and *NOHELP* in *getpar( )* calls meanwhile.)

4. Compile the application code, linking with the menu library and libtermcap.a. For example,

```
cc mycode.c -o mycode -lyamm -ltermcap
```

The resulting executable should display the menus as configured in your configuration file and allow you to execute the application functions. Once everything is working, you may want to change *config_name* and keep the file in a different place or keep the file contents exclusively in the code.

5. AN EXAMPLE

This section contains a detailed step-by-step implementation of a program using the menu package.

1. First create menuconfig:

```
# This is the menu configuration file. Each menu consists of MENU with
# the menu name, a number of menu entries, and an END; a menu entry is
# made up of the text to display for the menu selection and the
# function or submenu name, as appropriate. Terminal-specific
# sequences and keystroke-function mappings start with the TTY keyword
# and end with END. See the programmer documentation for more
# specifics. Comments should be prefaced with a #; all such text is
# ignored. Indentation may be used to illustrate menu structure. This
# text may also be put into the code. SCCS yamm_prog_guide 4.22 - 7/12/88

MENU mainmenu
Info/information     # function
Two Legs/twolegs     # submenu
Four Legs/fourlegs   # submenu
END

MENU twolegs
Cockatiel/bird
Parakeet/bird
```

```
Pigeon/bird
END

MENU fourlegs
Alsatian/dog
Collie/dog
German Shepherd/dog
END
```

2. Add in the call to *menu_start( )*, defining the menu config filename and menu config, function table, local (re)initialization routines, and logging file pointer. Make sure you include menuapp.h where necessary. Compile the code, linking with the menu (-lyamm) library and libtermcap.a (-ltermcap).

   Here is the code from the demo program contained in demo.c:

```
#ifndef lint
static char sccsid[]="@(#)yamm_prog_guide      4.22 7/12/88";
#endif lint

#include <ctype.h>
#include <stdio.h>
#include <strings.h>
#include "menuapp.h"

int information(),dog(),bird();

char *config_name = "./menuconfig";
char *config[] = {
   "MENU mainmenu","Information/information","Two Legs/twolegs",
      "Four Legs/fourlegs","END",
   "MENU twolegs","Cockatiel/bird","Parakeet/bird","Pigeon/bird","END",
   "MENU fourlegs","Alsatian/dog","Collie/dog","German Shepherd/dog","END",
   "/keep last"
};

struct func_name_pair func_table[]={
   {information,"information"},
   {dog,"dog"},
   {bird,"bird"},
   {0,"/keep last"},
};

main()
{
   FILE *logfile;
   logfile = fopen("demo.log","w");
   (void)menu_start(config_name,config,func_table,(int (*)())NULL,
      (int (*)())NULL,logfile);
   (void)fclose(logfile);
}

information()
{
   (void)printf("This is the Pets pet registry program.  ");
```

```
    (void)printf("Since this is for ");
    (void)printf("menu system demonstration only, no information is ");
    (void)printf("actually saved.  You are now at the top of a ");
    (void)printf("short menu tree.  By selecting menu entries ");
    (void)printf("(use the help key for more info) you can ");
    (void)printf("descend the tree in a manner compatible with your ");
    (void)printf("particular animal.  ");
    (void)printf("When you've reached the bottom of ");
    (void)printf("the menu tree, the function selected will display a ");
    (void)printf("parameter ");
    (void)printf("entry screen.  Feel free to move about the screen ");
    (void)printf("(use the help key for specifics) ");
    (void)printf("and fill in values.  If the program erases something ");
    (void)printf("immediately after you've typed it, the input is ");
    (void)printf("illegal.  ");
    (void)printf("Type the end-data-entry key ");
    (void)printf("when you're done entering parameter values.  Note that ");
    (void)printf("the program may refuse to leave if some parameter ");
    (void)printf("value is required and unspecified.  ");
    (void)printf("When you've reached the bottom of ");
    (void)printf("the menu tree, the function selected will display a ");
    (void)printf("parameter ");
    (void)printf("entry screen.  Feel free to move about the screen ");
    (void)printf("(use the help key for specifics) ");
    (void)printf("and fill in values.  If the program erases something ");
    (void)printf("immediately after you've typed it, the input is ");
    (void)printf("illegal.  ");
    (void)printf("Type the end-data-entry key ");
    (void)printf("when you're done entering parameter values.  Note that ");
    (void)printf("the program may refuse to leave if some parameter ");
    (void)printf("value is required and unspecified.  ");
}

/*-------------------------------------------------------------------
This function demonstrates the basic structure.  Note that some values
are statics.  These are initialized to default values the first time, and
then when this function is called in the future, the last values used are
the defaults.  Pet's name goes into name (up to 20 characters), and is
required; there is also a default.  Foods takes up to 5 values, stored
starting at location foods and every 20 bytes thereafter; the number of
foods selected is returned in food_count.  Eye color is simply a string,
stored in eyes.  The next two are boolean (YES or NO); note that booleans
should always be initialized.  The weights parameter illustrates how to
request input of a matrix (in this case, a weights matrix for your dog).
Note the use of the GROUP keyword (to enable table formation); this is
somewhat awkward but very simple and flexible.  The getpar request
displays the menu and returns with the values, which in this example, are
ignored.
-------------------------------------------------------------------*/
dog()
{
    int chk_dog(),help_dog(),food_count;
    char name[25],foods[20*5];
    static double size[2];
```

```
    static char eyes[20];
    static int license=0,plays=0,first_time=1;
    static float wgts[9];
    struct param params[9];
    (void)strcpy(name,"Rover");
    if (first_time) {
       (void)strcpy(eyes,"brown");
       first_time=0;
    }
    (void)def(params,0,"Pet's name",STRING,name,WID,20,LINE,1,DEFAULT,REQ,
       END);
    (void)def(params,1,"Favorite food(s)",STRING,foods,WID,15,INCR,20,
       COUNT,&food_count,LINE,1,COL,40,DUP,-5,END);
    (void)def(params,2,"Eye color",STRING,eyes,WID,10,LINE,3,DEFAULT,REQ,
       END);
    (void)def(params,3,"Licensed?",BOOL,&license,LINE,5,DEFAULT,END);
    (void)def(params,4,"Likes to chase cars?",BOOL,&plays,LINE,7,DEFAULT,
       END);
    (void)def(params,5,"Ht, Length (fractional) ?",DOUBLE,size,
       WID,10,LINE,9,DUP,2,REQ,END);
    (void)def(params,6,"Weights",FLOAT,wgts,INCR,sizeof(float)*3,LINE,7,
       COL,50,DUP,3,GROUP,0,ENTRYLINE,8,ENTRYCOL,40,END);
    (void)def(params,7,"Weights",FLOAT,wgts+1,INCR,sizeof(float)*3,LINE,7,
       COL,50,DUP,3,GROUP,0,ENTRYLINE,8,ENTRYCOL,50,END);
    (void)def(params,8,"Weights",FLOAT,wgts+2,INCR,sizeof(float)*3,LINE,7,
       COL,50,DUP,3,GROUP,0,ENTRYLINE,8,ENTRYCOL,60,END);
    if (getpar(params,9,chk_dog,help_dog) == -1) return;
/* continue with rest of program here - there isn't any in these examples */
    (void)printf("%f %f0,size[0],size[1]);
}


/*-----------------------------------------------------------------------
This is the error-checking routine for dog() above, which is very
straightforward.  Number is the number of the parameter, and value is the
value specified (a union of three differently typed variables).  If this
returns a non-zero value, getpar will erase what the user typed, indicating
that the input was erroneous.
-----------------------------------------------------------------------*/
/*ARGSUSED*/
chk_dog(params,number,value)
struct param *params;
int number;
union {int i; double f; char *s;} value;
{
    int error;
    switch (number) {
       case 0: error = alphawhite(value.s); break;
       case 1: error = alphawhite(value.s); break;
       case 2:
          if ((error=color(value.s))!=0) {
             (void)printf("Unknown color.  Use blue, gray, green ");
             (void)printf("or brown.");
          }
```

```
            break;
        default: error = 0; break;
    }
    return(error);
}

/*---------------------------------------------------------------------
This is the help routine for the dog() function above. When the user is
entering parameter values and presses the help key, getpar passes the
number of the parameter to this routine which prints the information
desired. Also, the help function is called by getpar immediately on startup
with a parameter value of -1 in case application has some specific prompt
for user.
---------------------------------------------------------------------*/
/*ARGSUSED*/
help_dog(params,number)
struct param *params;
int number;
{
    switch (number) {
        case -1: (void)printf("Welcome to the Dog menu"); break;
        case 0: (void)printf("Name of dog; the default is Rover"); break;
        case 1: (void)printf("Favorite foods"); break;
        case 2: (void)printf("Eye color"); break;
        case 3: (void)printf("Is your dog licensed?"); break;
        case 4:
            (void)printf("Do its ears perk up whenever it hears a VW?");
            break;
    }
}

/*---------------------------------------------------------------------
This is somewhat more complicated than the previous function. Parameter
2, and parameters 3 and 4 are mutually exclusive. (If a bird has its
wings clipped, it cannot fly; if it can fly, its wings are not clipped.)
This is enforced by chk_bird(). Also, if parameter 3 is specified, then
parameter 4 must be, and vice versa; hence, they are specified as
belonging to group 0 instead of NOGRP. This is enforced by the getpar
routine. The first parameter illustrates the use of legal-value
sets. Finally, if the third name is selected, the program dynamically
creates another similar menu below the current one and goes into it.
This one contains no information option.
---------------------------------------------------------------------*/
bird()
{
    int clipped=0,dist,chk_bird(),help_bird();
    float height;
    static char name[25];
    static int first_time = 1;
    char units[20];
    struct param params[5];
    static char *poss_values[] = {"rover","king","ohnooo!"};
    static char *newconfig[] = {
        "MENU mainmenu","Two Legs/twolegs","Four Legs/fourlegs","END",
```

```
        "MENU twolegs","Cockatiel/dog","Parakeet/dog","Pigeon/dog","END",
        "MENU fourlegs","Alsatian/bird","Collie/bird","German Shepherd/bird",
            "END","/keep last"
    };
    if (first_time) {
        (void)strcpy(name,"king");
        first_time = 0;
    }
    (void)def(params,0,"Pet's name",STRING,name,WID,20,LINE,2,REQ,ENUM,3,
        (unsigned char *)poss_values,DEFAULT,END);
    (void)def(params,1,"Height (fractional inches)",FLOAT,&height,WID,6,
        LINE,4,COL,40,END);
    (void)def(params,2,"Wings clipped?",BOOL,&clipped,LINE,4,DEFAULT,END);
    (void)def(params,3,"Maximum flying distance (integer)",INT,&dist,WID,4,
        LINE,8,GROUP,0,END);
    (void)def(params,4,"Units",STRING,units,WID,10,LINE,8,COL,40,GROUP,0,
        END);
    if (getpar(params,5,chk_bird,help_bird) == -1) return;
    if (strcmp(name,"ohnooo!") == 0) {
        (void)printf("Ohnooo!");
        (void)create_new_subtree((char *)NULL,newconfig,func_table,
            (int (*)())NULL,(int (*)())NULL);
    }
    else (void)printf("What kind of name is
/* continue with rest of program here - there isn't any in these examples */
}

/*-----------------------------------------------------------------------
Note in this function the handling at cases 2 through 4.  If parameter 2
becomes true (wings clipped), specified values for parameters 3 and 4 are
removed.  Similarly, if parameter 3 or 4 is given a value, then parameter
2 is cleared.  This is done using unspecify, which removes the value for
the specified parameter, and new_value, which tells the menu package that
the error routine has changed a value and it should be redisplayed.
-----------------------------------------------------------------------*/
/*ARGSUSED*/
chk_bird(params,number,value)
struct param *params;
int number;
union {int i; double f; char *s;} value;
{
    int error,i,pr_msg;
    char locbuf[30];
    switch (number) {
        case 0: error = alphawhite(value.s); break;
        case 1:
            error = (value.f<2. || value.f>15.? -1:0);
            if (error == -1)
                (void)printf("Birds must be between 2 and 15 inches tall.");
            break;
        case 2:
            error = 0;
            pr_msg = 0;
            if (value.i && !unspecified(3)) {
```

```
            (void)unspecify(params,3);
            pr_msg=1;
         }
         if (value.i && !unspecified(4)) {
            (void)unspecify(params,4);
            pr_msg=1;
         }
         if (pr_msg)
            (void)printf("Changing flying distance to be undefined.");
         break;
      case 3:
         error = (value.i<0? -1:0);
         if (error == 0 && !unspecified(2) && *((int *)params[2].value)) {
            *((int *)params[2].value) = 0; /* or clipped=0; */
            (void)new_value(params,2);
            (void)printf("Changing wings to
         }
         break;
      case 4:
         error = 0;
         (void)strcpy(locbuf,value.s);
         for (i=0;i<strlen(locbuf);i++)
            if (isupper(locbuf[i])) locbuf[i]+=32;
         if (strcmp(locbuf,"inches")!=0 && strcmp(locbuf,"feet")!=0 &&
            strcmp(locbuf,"yards")!=0 && strcmp(locbuf,"miles")!=0)
               error = -1;
         if (error == -1)
            (void)printf("Please use inches, feet, yards, or miles.");
         else if (!unspecified(2) && *((int *)params[2].value)) {
            *((int *)params[2].value) = 0; /* or clipped=0; */
            (void)new_value(params,2);
            (void)printf("Changing wings to
         }
         break;
      default: error = 0; break;
   }
   return(error);
}

/*ARGSUSED*/
help_bird(params,number)
struct param *params;
int number;
{
   switch (number) {
      case 0: (void)printf("Name of bird."); break;
      case 1: (void)printf("Height, between 2 and 15 inches"); break;
      case 2: (void)printf("Are wings clipped?"); break;
      case 3:
         (void)printf("Maximum flying distance.  ");
         (void)printf("If this is specified, distance units must also ");
         (void)printf("be specified and vice versa.");
         break;
      case 4:
```

```
            (void)printf("Distance units.  ");
            (void)printf("If this is specified, maximum flying distance ");
            (void)printf("must also be specified and vice versa.");
            break;
    }
}

/*---------------------------------------------------------------------
Error checking routines:
alphawhite(buf) returns 0 iff buf is only letters and white space
color(buf) returns 0 iff buf is a valid color
---------------------------------------------------------------------*/
alphawhite(buf)
char *buf;
{
    int i;
    for (i=0;i<strlen(buf);i++)
        if (!isalpha(buf[i]) && !isspace(buf[i])) return(-1);
    return(0);
}

color(buf)
char *buf;
{
    int i;
    char locbuf[30];
    (void)strcpy(locbuf,buf);
    for (i=0;i<strlen(locbuf);i++) if (isupper(locbuf[i])) locbuf[i]+=32;
    if (strcmp(locbuf,"brown")==0 || strcmp(locbuf,"blue")==0 ||
        strcmp(locbuf,"green")==0 || strcmp(locbuf,"gray")==0) return(0);
    else return(-1);
}
```

6. INTERNALS

There are a few internals issues that a programmer should be aware of.

Yamm changes the handling of stdout and stderr in order that program output can be intercepted and properly placed in the menu windows. In the unlikely event of a bug in the output handling routines, traces and other output statements might not function correctly. To bypass this processing, the file pointer *mm_termfp* may be referenced using *extern*; this pointer directly accesses the screen.

Also, Yamm starts up a subprocess (*menuwatch*) and allocates a semaphore in order to properly handle application I/O. Otherwise application I/O would be limited to 4096 bytes (the size of the buffered I/O buffer) and flushed only on application completion. The subprocess waits for application I/O and then signals Yamm to print it in the proper place. Normally, when an application dies, the subprocess will also die, freeing the semaphore in the process. However, if the subprocess is killed, the semaphore may not get removed; when enough semaphores build up, Yamm will not start properly. To avoid this, the user should never kill the subprocess explicitly. If semaphores are left around, *ipcrm(1)* may be used to remove them.

In order to dump the screen, Yamm creates a temporary file in the user's current directory and then attempts to troff this file using the pipeline *tbl | ptroff -ms*. If this succeeds, the file is then deleted; otherwise the file remains. The user may specify an alternative command (or no command) by setting the environment variable *MENU_HARDCOPY_CMD*. For example, the command *setenv MENU_HARDCOPY_CMD 'tbl %s | ptroff -ms'* would print hardcopies without deleting the originals.

The names of temporary files are determined by the day and time of the screen dump; e.g., *menupic1091252* was dumped Monday (1), at time 9:12:52.

Appendix H – CIPETOOL

Subroutine: **main(argc, argv)**

File: main.c

Input: **int argc**
char *argv[]

Output: none

Function: This is the main program for Cipetool. It calls some initialization routines and then calls window_main_loop which is the Sunview dispatcher. Window_main_loop will return when the user clicks on the "quit" button. Currently, **main** does not use any arguments.

Suggested Modification:
An argument should be used to indicate which version of CIPE to run. This would make it convenient to use a new version of CIPE.

Subroutine: **Notify_value wait3_handler(me, pid, status, rusage)**

File: support.c

Input: **int *me**
int pid
union wait *status
struct rusage *rusage

Output: none

Function: This function will be called by the dispatcher if the CIPE command line interpreter dies. When this happens, Cipetool will print an error message and exit.

Suggested Modification:
none

Subroutine: **button_pressed(item, event)**

File: support.c

Input: **Panel_item item**
Event *event

Output: none

Function: This routine will be called whenever the user clicks on one of the buttons. It checks to see which button was pressed and calls the appropriate routine to handle it.

Suggested Modification:
none

Subroutine: **type_text(window, event, arg)**

File: support.c

Input: **Window window**
Event *event
caddr_t arg

Output: none

Function: This routine will be called whenever the user types in the text window connected to CIPE. It will send the character that the user typed to CIPE.

Suggested Modification:
This is not dealt with correctly right now. You can send information to CIPE this way, but Cipetool will get confused when CIPE responds. This should be fixed so that the user can use Cipetool both as an icon-based tool and a Cli-based tool.

Subroutine: **list_functions(event)**

File: support.c

Input: **Event *event**

Output: none

Function: This is a very short routine that gets called whenever the user clicks on the "functions" button. It does a menu_show of the functions menu.

Suggested Modification:
This routine really doesn't need to exist. The call to menu_show could be moved up into **button_pressed.**

Subroutine: **Notify_value frame_event_handler(win, event, arg)**

File: windows.c

Input: **Window win**
Event *event
caddr_t arg

Output: none

Function: This is the event handler for the Cipetool frame.

Suggested Modification:
none

Subroutine: **init_windows( )**

File: windows.c

Input: none

Output: none

Function: This routine is called to initialize all of the startup windows. It creates all of the buttons as well.

Suggested Modification:
none

Subroutine: **struct symtab_type *window_under_cursor(x, y)**

File: windows.c

Input: **int x**
int y

Output: none

Function: This routine returns a pointer to a symbol which is the window under the cursor. Normally this will be an image window or a function window.

Suggested Modification:
none

Subroutine: **connect_wins(start_win, start_x, start_y, end_win, end_x, end_y)**

File: windows.c

Input: **Window start_win**
int start_x
int start_y
Window end_win
int end_x
int end_y

Output: none

Function: This routine will draw a line between two windows.

Suggested Modification:
This routine should be called whenever one of the the windows is moved. It is only called once, currently, but this means that you have lines drawn to nowhere if you move a window.

Subroutine: **void draw_main_canvas(canvas, pixwin, repaint_area)**

File: windows.c

Input: **Canvas canvas**
Pixwin *pixwin
Rectlist *repaint_area

Output: none

Function: Draw_main_canvas draws the frame buffer area onto the Cipetool canvas.

Suggested Modification:
none

Subroutine: **in_frame_buffer**

File: windows.c

Input: **int x**
int y

Output: none

Function: This routine will return a 1 if the x and y positions passed are within the frame buffer area of the Cipetool canvas and a 0 otherwise. It is assumed that x and y are relative to that canvas.

Suggested Modification:
none

Subroutine: **init_symtab( )**

File: symtab.c

Input: none

Output: none

Function: Init_symtab should be called before any other symbol table routine is called. It initializes the hash table of symbols. The hash table has header cells to make insertion and deletion from the symbol table easier.

Suggested Modification:
none

Subroutine: **hash(name)**

File: symtab.c

Input: **char *name**

Output: none

Function: This routine returns the bin number where "name" should be stored or retrieved. The current hashing functions are the sum of the squares of the ASCII values of "name."

Suggested Modification:
none

Subroutine: **struct symtab_type *add_symbol(name)**

File: symtab.c

Input: **char *name**

Output: none

Function: This routine creates a new symbol table entry for name and returns a pointer to that entry. It initializes some of the fields in the entry.

Suggested Modification:
none

Subroutine: **struct symtab_type *find_symbol(name)**

File: symtab.c

Input: **char *name**

Output: none

Function: Given the name of a symbol, this function returns a pointer to the entry for that symbol. If there is no such symbol then a NULL is returned.

Suggested Modification:
none

Subroutine: **delete_symbol(name)**

File: symtab.c

Input: **char *name**

Output:

Function: Delete_symbol will delete the symbol **name** from the symbol table. If there are multiple entries for **name**, only the first one will be removed. The current implementation does not ever have multiple entries, but this assumption is not made in any of the symbol table routines.

Suggested Modification:
none

Subroutine: **char *unique_constant_symbol()**

File: symtab.c

Input: none

Output: none

Function: This function returns a string which is guaranteed not to exist in the symbol table. Currently, this is done by appending a number to the string "constant." The number is a static that gets incremented on each call to this routine. A check is made to see if this symbol is in the symbol table. If it is, then the static will be incremented until this check fails.

Suggested Modification:
none

Subroutine: **char *unique_image_symbol( )**

File: symtab.c

Input: none

Output: none

Function: This routine functions like **unique_constant_symbol** but it appends the number to the string "image."

Suggested Modification:
none

Subroutine: **struct param_type *new_params( )**

File: symtab.c

Input: none

Output: none

Function: This function is called by the parser for the functions file. It gets called whenever the parser encounters an argument list. It initializes the empty parameter structure and returns it.

Suggested Modification:
none

Subroutine: **struct param_type *add_param(params)**

File: symtab.c

Input: **struct param_type *params**

Output: **struct param_type *params**

Function: This routine adds another parameter to the end of params and returns a pointer to this new entry. It will be called by the parser to add another argument to a parameter list. This routine does **not** fill in the parameter structure. This needs to be done by the caller.

Suggested Modification:
none

Subroutine: **yyparse( )**

File: rc.y

Input: none

Output: none

Function: This routine is generated by the UNIX utility YACC. It is the parser used to parse the file declaring all of the available application functions and their arguments and return values. A LALR description of the language is in the file rc.y.

Suggested Modification:
This version of the parser is incompatible with Alan Mazer's version now. This implies that some of the function definition files used by Alan will not work with Cipetool. This needs to be fixed.

Subroutine: **yyerror(s)**

File: rc.y

Input: **char *s**

Output: none

Function: This function is called by the parser whenever an error is encountered while parsing the function definition file. It prints the string on **stderr** along with the line number of the offending command.

Suggested Modification:
none

Subroutine: **yylex**
File: rclex.l
Input: none
Output: none
Function: This is the lexical analyzer for the function definition file. It is called by yyparse whenever a new token is desired.
Suggested Modification:
none

Subroutine: **print_tabs(level)**
File: name_list.c
Input: **int level**
Output: none
Function: This routine is mainly for debugging. It is called by print_files to print out **level** tabs.
Suggested Modification:
none

Subroutine: **print_files(files, level)**
File: name_list.c
Input: **struct name_list_type *files**
int level
Output: none
Function: This routine is used for debugging. It prints out **files** with each directory level being indented one more tab than the previous level.
Suggested Modification:
none

Subroutine: **char *basename(name)**
File: name_list.c
Input: **char *name**
Output: none
Function: This routine returns a pointer to the basename of the filename it is called with. A files basename is the last part of the filename after any '/'s. For instance, basename("/ufs/images/girl.bw") is "girl.bw."
Suggested Modification:
This routine returns a pointer into the argument, which can lead to some strange results sometimes. It might be better for this to be changed.

Subroutine: **struct name_list_type *get_image_files(filename)**
File: name_list.c
Input: **char *filename**
Output: none
Function: This function reads in all of the image files in the directory **filename** or its children. It reads each file in the directory to see if it has a corresponding image header file. If it does, then it adds it to its list of image files.
Suggested Modification:
none

Subroutine: **length_name_list(list)**
File: name_list.c
Input: **struct name_list_type *list**
Output: none
Function: This function returns the number of elements in a name list.
Suggested Modification:
none

Subroutine: **print_name_list(list)**
File: name_list.c
Input: **struct_name_list_type *list**
Output: none
Function: This function prints all of the elements of a name list.
Suggested Modification:
none

Subroutine: **sort_name_list(list)**
File: name_list.c
Input: **struct name_list_type *list**
Output: none
Function: This routine uses a quick sort to sort the members of a name list in ASCII order. Note that this routine sorts by moving pointers in the list around. So any pointers into the middle of the name list before the sort will probably be invalid after the sort.
Suggested Modification:
none

Subroutine: **mystrcmp(p1, p2)**
File: name_list.c
Input: **struct name_list_type **p1**
struct name_list_type **p2
Output: none
Function: This routine just contains a call to **strcmp** with the name field of the name_list. It is called by the quicksort in **sort_name_list.**
Suggested Modification:
none

Subroutine: **add_function(name)**
File: name_list.c
Input: **char *name**
Output: none
Function: This routine adds a function to the list of functions in the global **functions_list.** This list is used to create the functions menu.
Suggested Modification:
none

Subroutine: **init_images( )**
File: images.c
Input: none
Output: none
Function: This routine is responsible for doing any initialization of image structures. It should be called before any of the other functions in images.c. For now, it just initializes the global variable **image_canvases,** which is a list of all of the canvases associated with image icons.
Suggested Modification:
none

Subroutine: **load_images(image_menu, files)**
File: images.c
Input: **Menu image_menu**
struct name_list_type *files
Output:
Function: This routine takes a name list of files and builds a menu out of it. If the name list has several levels, i.e., it represents several directory levels, then the menu will have several levels with pullright menus representing the different levels.
Suggested Modification:
none

Subroutine: **image_selected(menu, item)**

File: images.c

Input: **Menu menu**
Menu_item item

Output: none

Function: This routine will be called by the dispatcher whenever the user selects an image from the image menu. It will figure out which image was selected, create a symbol name for that image, and send a command to CIPE to read in that image. Once it has done this it calls **draw_image** to draw an iconic representation of this image.

Suggested Modification:
none

Subroutine: **struct image_header_type *get_image_header(filename)**

File: images.c

Input: **char *filename**

Output: none

Function: This routine will return the CIPE header for those files that CIPE knows about. For now it looks for a file that ends in .hdr. It first appends .hdr to the filename, and if this fails it searches for basename(filename).hdr, where basename is the filename without the suffix. For example, if the filename is cheryl.red and there is no cheryl.red.hdr then cheryl.hdr will be opened if it exists.

The header is ASCII so it is humanly readable and writable. If the .hdr file does not contain a valid CIPE header, or does not exist, then a NULL pointer is returned.

Suggested Modification:
In the future, this routine will probably be replaced with one that searches a database for the headers.

Subroutine: **skip_blanks(fp)**

File: images.c

Input: **FILE *fp**

Output: none

Function: This routine is not called by anyone.

Suggested Modification:
Get_image_header should call this routine to allow spaces between the key words.

Subroutine: **put_image_header(filename, header)**
File: images.c
Input: **char *filename**
struct image_header_type *header
Output: none
Function: This routine will return the CIPE header for those files that CIPE knows about. For now, it looks for a file that has .hdr appended to the original file name and reads the header out of there. The header is ASCII so it is humanly readable and writable. If the .hdr file does not contain a valid CIPE header, or does not exist, then a NULL pointer is returned.
Suggested Modification:
none

Subroutine: **show_image(name)**
File: images.c
Input: **char *name**
Output: none
Function: This routine is not called by anyone any more. It displays a single band of an image on the screen. This routine was used during debugging. It still exists because it may be of some use at some later date.
Suggested Modification:
none

Subroutine: **function_selected(menu, item)**
File: functions.c
Input: **Menu menu**
Menu_item item
Output: none
Function: This function is called by the dispatcher when the user selects a function from the functions menu. It figures out which function was selected and then calls **draw_function** to draw an icon of it.
Suggested Modification:
none

Subroutine: **init_functions( )**
File: functions.c
Input: none
Output: none
Function: This routine is responsible for initializing the functions menu. It calls routines to read and parse the users .cipetoolrc so that it knows what functions are available.
Suggested Modification:
none

Subroutine: **build_functions_menu( )**

File: functions.c

Input: none

Output: none

Function: Once the list of functions as been created by the parser, **build_functions_menu** will create the menu to hold them all.

Suggested Modification:
The current implementation may look unusual if there are over 50 functions.

Subroutine: **Notify_value function_event_proc(win, event, arg)**

File: functions.c

Input: **Window win**
Event *event
caddr_t arg

Output: none

Function: This routine will be called whenever the user clicks the mouse while over a function icon. If the button is the left button, then this event will signal the beginning or end of the connection of two functions. If it is the beginning, it sets a global indicating this, otherwise it draws a line between the two windows.

Suggested Modification:
This routine should never be called. All windows now have the same event handler, namely **button_in_window**.

Subroutine: **draw_function(name)**

File: functions.c

Input: **char *name**

Output: none

Function: **Draw_function** draws the icon for a specific function. It creates the frame and canvas for it and then draws in the boxes and labels indicating parameters and return values.

Suggested Modification:
none

Subroutine: **struct param_type *nearest_parameter(symbol, xpos, ypos)**

File: functions.c

Input: **struct symtab_type *symbol**
int xpos
int ypos

Output: none

Function: Given a symbol and a mouse position, this routine will return a pointer to the nearest parameter.

Suggested Modification:
none

Subroutine: **type_of_result(s)**

File: functions.c

Input: **char *s**

Output: none

Function: This routine is called after a function has been sent to CIPE and we are waiting for the results. It determines the type of the result and draws the corresponding icon. For instance if "filter" is called, it will return an image. **Type_of_result** will determine this and draw an icon for that image.

Suggested Modification:
This routine needs to be fixed so that it does not get confused so easily. It also needs to take care of other types besides images. This may be a lower priority, because most of our tools return images.

Subroutine: **error(string)**

File: error.c

Input: **char *string**

Output: none

Function: This function pops up a window with a given message on it (usually an error message) for a given number of seconds and then removes it. The number of seconds that the message stays up is given by the global **error_on_screen**. This is implemented rather strangely. The window is displayed and an alarm is set to go off after so many seconds. That alarm will call **error_off** which will get rid of the window.

Suggested Modification:
In Sunview I in SunOS 4.0, there is a new class of windows called alerts. **Error** should be rewritten to use this class.

Subroutine: **Notify_value error_off(me, which)**

File: error.c

Input: **Notify_client me**
int which

Output: none

Function: This routine deletes the error window. It is used in conjunction with **error** above.

Suggested Modification:
none

Subroutine: **Notify_value button_in_window(win, event, arg)**

File: draw_image.c

Input: **Window win**
Event *event
caddr_t arg

Output: none

Function: This function is called whenever the mouse is clicked in a window. It is responsible for determining the type of the window (image, function, or constant) and doing whatever is necessary. Most of the time a left click will be used to indicate the beginning or end of a line between a function and its arguments.

Suggested Modification:
none

Subroutine: **draw_image(image_name, image_header)**

File: draw_image.c

Input: **char *image_name**
struct image_header_type *image_header

Output: none

Function: This routine will draw the iconic representation of an image. The icon is proportional to the height and width of the image.

Suggested Modification:
Currently, only two dimensional images are supported. The current plan is to have a scrollbar-like object that allows you to select particular bands. Currently, selecting a subsection of the image is not supported either. This will be done by clicking a mouse button at the desired upper left corner and dragging to the desired lower right corner.

Subroutine: **which_band_proc(item, value, event)**

File: draw_image.c

Input: **Panel_item item**
int value
Event *event

Output: none

Function: This routine determines which band has been selected using the scrollbar. It is not called at the current time but will probably be called when multiband images are supported.

Suggested Modification:
none

Subroutine: **constant_window( )**

File: constant.c

Input: none

Output: none

Function: This routine creates a window into which the user can enter a constant. When the number is entered, the window will change shape to exactly fit the constant. If the constant is a string, then it should be surrounded by double quotes.

Suggested Modification:
none

Subroutine: **Panel_setting constant_entered(item, event)**

File: constant.c

Input: **Panel_item item**
Event *event

Output: none

Function: When the user types a key in a constant window, this function gets called by the dispatcher. It determines if the key is valid and echos it in the window if it is. It also checks for the end of the input (a carriage return) and resizes the window appropriately.

Suggested Modification:
none

Subroutine: **array_window( )**

File: array.c

Input: none

Output: none

Function: This routine creates a window into which the user can enter an array. When the elements are entered, the window will change shape to exactly fit the array.

Suggested Modification:
There is no way to enter two-dimensional arrays. The correct method has not been determined.

Subroutine: **Panel_setting array_entered(item, event)**

File: array.c

Input: **Panel_item item**
Event *event

Output: none

Function: When the user types a key in an array window this function gets called by the dispatcher. It determines if the key is valid and echos it in the window if it is. It also checks for the end of the input (a carriage return) and resizes the window appropriately.

Suggested Modification:
none

Subroutine: **get_results_routine(func)**

File: cipe_comm.c

Input: **int (*func)()**

Output: none

Function: This routine sets up the function that will be called whenever the results from a command are received. The function will be called with a ptr to a string containing the line received. When the command is finished (i.e., the '> ' prompt is received) the function will be called once more with a NULL pointer. The function should use this as a signal to clean up.

Suggested Modification:
none

Subroutine: **Notify_value read_pipe(me, fd)**

File: cipe_comm.c

Input: **int *me**
int fd

Output: none

Function: This function is called by the dispatcher whenever there is data available on the pipe connected to CIPE. It reads the pipe, echos it to the CIPE window and calls any functions that have been set to handle the input.

Suggested Modification:
Read_pipe gets confused when CIPE returns unsolicited input. Currently, any output from an application will cause it to lose track of its current state. This should be fixed.

Subroutine: **sub_proc(proc_name, argv, send, receive)**

File: cipe_comm.c

Input: **char *proc_name**
char *argv[]
int *send
int *receive

Output: none

Function: This function will fork a process with two pipes connected to it. One is attached to the stdin of the forked process and one to the stdout. These two fd's will be returned in **send** (attached to stdin) and **receive** (attached to stdout). The process id (pid) of the forked process will be returned. If something went wrong (could not fork process or could not exec) then -1 is returned.

Suggested Modification:
none

Subroutine: **send_string(s)**
File: cipe_comm.c
Input: **char *s**
Output: none
Function: This procedure is used to send a string to CIPE.
Suggested Modification:
Calls to this routine should probably be replaced with calls to **fputs**.

Subroutine: **init_comm( )**
File: cipe_comm.c
Input: none
Output: none
Function: This routine will initialize communication with CIPE. It creates a child remote shell (rsh) process to the hypercube host to run CIPE and sets up file pointers to communicate with it. File descriptors could have been used, but they are not nearly as convenient.
Suggested Modification:
none

Appendix I – CIPE INCLUDE FILES

```
/* cipedict version 2.1 7/25/88 */

function filter(input "In", input "Box width", input "Box height", input "Wgts",
        input "Loadtype")
   returns "Out" pathname "/ufs/cipe/appl/cp/cpfilter"
function geom(input "In", input "Tiepoints filename")
   returns "Out" pathname "/ufs/cipe/appl/cp/cpgeom"
function size(input "In", input "Line zoom", input "Sample zoom")
   returns "Out" pathname "/ufs/cipe/appl/cp/cpsize"
function clear()
   returns "Out" pathname "/ufs/cipe/appl/cp/clear"
```

```
/* builtin.h version 2.1, 7/25/88 */

#ifndef BUILTIN_H
#define BUILTIN_H

/* function name definition */

#define LESS_THAN_FUNCTION              1
#define LESS_THAN_EQUAL_FUNCTION        2
#define EQUAL_FUNCTION                  3
#define NOT_EQUAL_FUNCTION              4
#define GREATER_THAN_FUNCTION           5
#define GREATER_THAN_EQUAL_FUNCTION6
#define AND_FUNCTION                    7
#define OR_FUNCTION                     8
#define BITAND_FUNCTION                 9
#define BITOR_FUNCTION                  10
#define BITXOR_FUNCTION                 11
#define PLUS_FUNCTION                   12
#define MINUS_FUNCTION                  13
#define MULT_FUNCTION                   14
#define DIV_FUNCTION                    15
#endif BUILTIN_H
```

```
/* cipe.h version 2.2, 8/23/88 */

#ifndef CIPE_H
#define CIPE_H

struct image_h {
   char    header_indicator[5];
   int     datatype,
           nl,
           ns,
           nb,
           offset;              /* offset in bytes where data starts. */
   float   swl,
           ewl;
                                /* used to skip headers. */
   unsigned char   reserved[32];
   struct image_header_type *next;      /* for use when images */
                                        /* are in a linked list. */
};


#define CIPENAMESIZE 80
typedef  char cipe_file_name[CIPENAMESIZE];
typedef  char cipe_sym_name[CIPENAMESIZE];

#include "symbol.h"
#include "funclib.h"

   /* general constants */

#ifndef TRUE
#define TRUE          1
#define FALSE         0
#endif TRUE

#define OFF    0
#define ON     1

#define IVAS_UNIT_NUM     0
#define RASTER_UNIT_NUM 1

extern int doc,nproc;

   /* system attributes */

struct attr_def {       /* ------------------------------------------- */
   char *name;          /* mods to this section must also be in cipeappl.h! */
   int value_type;      /* ------------------------------------------- */
   int value;
   int access;
};

#define COPROCESSOR_ATTR              0
#define CUBE_DIMENSION_ATTR           1
```

```
#define DISPLAY_DEVICE_ATTR          2
#define MOUSE_ATTR                   3
#define LOGGING_ATTR                 4
#define DEBUG_LEVEL_ATTR             5
#define LEX_TRACE_ATTR               6
#define PARSE_TRACE_ATTR             7
#define CODEGEN_TRACE_ATTR           8
#define EXEC_TRACE_ATTR              9
#define SYMTAB_TRACE_ATTR            10
#define FUNCTAB_TRACE_ATTR           11
#define CUBE_COMMAND_TRACE_ATTR      12
#define CUBE_DATA_TRACE_ATTR         13
#define CUBE_SYMBOL_TRACE_ATTR       14
#define APPL_TRACE_ATTR              15
#define MENU_MODE_ATTR               16
#define NUM_ATTRS                    17

extern struct attr_def cipe_attr[];

#define COPROCESSOR                  cipe_attr[COPROCESSOR_ATTR].value
#define CUBE_DIMENSION               cipe_attr[CUBE_DIMENSION_ATTR].value
#define DISPLAY_DEVICE               cipe_attr[DISPLAY_DEVICE_ATTR].value
#define MOUSE                        cipe_attr[MOUSE_ATTR].value
#define LOGGING                      cipe_attr[LOGGING_ATTR].value
#define DEBUG_LEVEL                  cipe_attr[DEBUG_LEVEL_ATTR].value
#define LEX_TRACE                    cipe_attr[LEX_TRACE_ATTR].value
#define PARSE_TRACE                  cipe_attr[PARSE_TRACE_ATTR].value
#define CODEGEN_TRACE                cipe_attr[CODEGEN_TRACE_ATTR].value
#define EXEC_TRACE                   cipe_attr[EXEC_TRACE_ATTR].value
#define SYMTAB_TRACE                 cipe_attr[SYMTAB_TRACE_ATTR].value
#define FUNCTAB_TRACE                cipe_attr[FUNCTAB_TRACE_ATTR].value
#define CUBE_COMMAND_TRACE           cipe_attr[CUBE_COMMAND_TRACE_ATTR].value
#define CUBE_DATA_TRACE              cipe_attr[CUBE_DATA_TRACE_ATTR].value
#define CUBE_SYMBOL_TRACE            cipe_attr[CUBE_SYMBOL_TRACE_ATTR].value
#define APPL_TRACE                   cipe_attr[APPL_TRACE_ATTR].value
#define MENU_MODE                    cipe_attr[MENU_MODE_ATTR].value

#define ALL                  0
#define INTERNAL             1

   /* keyword values */

struct keyword_value_def {
   char *name;
   int attribute;
};

#define NOCOPROCESSOR_KWD    0
#define CUBE_KWD             1
#define GAPP_KWD             2
#define NODISPLAYDEVICE_KWD  3
#define IVAS_KWD             4
#define RASTER_KWD           5
```

```
#define NUM_KEYWORD_VALUES 6

extern char *bools[];
extern struct keyword_value_def keyword_values[];

    /* session logging */

#define printf cipe_printf
#define fgets cipe_fgets
#define gets(a) cipe_fgets(a,80,stdin)
extern char *cipe_fgets();
extern int cipe_printf();
extern int session_log_num;

extern FILE *session_log;

    /* code segments */

struct code_seg_def {
    char *name;
    struct instruction *code;
    struct cipe_symbol *symbols;
    struct cipe_function *functions;
    unsigned int code_length,max_code_length;
    unsigned int stail,ftail,max_stail,max_ftail;
    unsigned char workspace;
    struct code_seg_def *previous;
};

extern struct code_seg_def *code_seg;

    /* setjmp/longjmp */

#include <setjmp.h>

extern jmp_buf orig_env;
#endif CIPE_H

    /* installation-dependent defs */

#define CIPE_DICTIONARY   "/spacely/ufs/cipe/include/cipedict"
```

```
/* cipeappl.h version 2.1, 7/25/88 */

#ifndef CIPEAPPL_H
#define CIPEAPPL_H

#include <local/menuapp.h>
#include "symbol.h"
#include "elt_data.h"

extern struct {
    char *name;
    int value_type;
    int value;
    int access;
} cipe_attr[];

#define COPROCESSOR_ATTR        0
#define CUBE_DIMENSION_ATTR     1
#define MOUSE_ATTR              3
#define DEBUG_LEVEL_ATTR        5
#define APPL_TRACE_ATTR         15
#define MENU_MODE_ATTR          16

#define COPROCESSOR             cipe_attr[COPROCESSOR_ATTR].value
#define CUBE_DIMENSION          cipe_attr[CUBE_DIMENSION_ATTR].value
#define MOUSE                   cipe_attr[MOUSE_ATTR].value
#define DEBUG_LEVEL             cipe_attr[DEBUG_LEVEL_ATTR].value
#define APPL_TRACE              cipe_attr[APPL_TRACE_ATTR].value
#define MENU_MODE               cipe_attr[MENU_MODE_ATTR].value

#define CIPENAMESIZE 80
typedef  char cipe_file_name[CIPENAMESIZE];
typedef  char cipe_sym_name[CIPENAMESIZE];

/* cli related definitions */
extern symbolnum cipe_docall_args[];
extern symbolnum cipe_docall_result;

#define NOK             0
#define OK              1
#define ARG(n)                  cipe_docall_args[n]
#define ARGPTR(n)       cipe_symbol_index_to_ptr(ARG(n))
#define ARGDATA(n)      CIPEDATA(ARGPTR(n))
#define RESULTP                 cipe_symbol_index_to_ptr(cipe_docall_result)

extern int doc,nproc;
#endif CIPEAPPL_H
```

```
/* codegen.h version 2.1, 7/25/88 */

#ifndef CODEGEN_H
#define CODEGEN_H

extern struct    code_seg_def *cipe_init_code_seg();
extern struct    instruction *cipe_add_br_placeholder();

extern int compilation_error;

#define MAX_ARGUMENTS  15

struct forval_def {
   struct instruction *br1_loc;
   symbolnum var,expr2;
   struct instruction *expr2_loc;
   symbolnum step;
};

struct ifval_def {
   symbolnum expr;
   struct instruction *br1_loc;
};

struct elseval_def {
   struct instruction *br_loc,*backpatch;
};

struct indexlist_def {
   unsigned int dim;
   symbolnum start,end;
   struct indexlist_def *next;
};

struct exprlist_def {
   unsigned int nexprs;
   symbolnum expr;
   struct exprlist_def *next;
};

struct arglist_def {
   unsigned int nargs;
   symbolnum arg;
   struct arglist_def *next;
};

struct location_def {
   symbolnum line;
   symbolnum sample;
};

struct var_name_def {
   char *name;
   char *bufpos;
```

```
};

struct attrspec_def {
    unsigned int attribute;
    char *bufpos;
};

struct singleval_def {
    symbolnum symbol;
    char *bufpos;
};

struct integer_def {
    int value;
    char *bufpos;
};

struct float_def {
    float value;
    char *bufpos;
};

struct string_def {
    char *value;
    char *bufpos;
};

struct keyword_def {
    unsigned int value;
    char *bufpos;
};

struct boolean_def {
    int value;
    char *bufpos;
};

#endif CODEGEN_H
```

```
/* cp_mon.h version 2.1, 7/25/88 */

/*
 * This file defines things used by the CP side of the CIPE
 * hypercube interface.
 */

#ifndef CP_MON_H
#define CP_MON_H

#include "cipe.h"

#define MAXDOC 8

extern unsigned char whereis;    /* location of data  (bit map)*/
extern int stail;          /* tail index of the symbol table */
extern int sindex;                 /* searched symbol index */
extern cipe_sym_name input,output;

extern int *bufmap_send;
extern int *bufmap_receive;
extern int doc,nproc;


extern struct cmdpacket cmd;

#ifdef UNDEF          /* no longer used */
/* send a command to the nodes */
/* CP_CMD(message,name,ackwanted); */
#define CP_CMD(m,n,a)        cmd.message=(m);strcpy(cmd.name,(n));cmd.ackwanted=(a);bcastcp(&cmd,sizeof(cı
#endif UNDEF

#endif CP_MON_H
```

```
/* elt_data.h version 2.1, 7/25/88 */

 * This file contains definitions for the data transfer functions in the
 * Hypercube node monitor
 */

#ifndef ELT_DATA_H
#define ELT_DATA_H

/* definitions for "simple" data distribution types
 * i.e. values for elt_symbol.loadtype
 */
#define NOT_DIST        0       /* data not in cube */
#define BCAST_DIST      1       /* broadcast - all nodes have a copy */
#define HORIZ_DIST      2       /* row major decomposition */
#define VERT_DIST       3       /* column major decomposition */
#define GRID_DIST       4       /* grid distribution */
#define CUSTOM_DIST     5       /* a custom distribution */

/* The following definitions used to limit amount downloaded in
   CUSTOM_DISTs at one time */

#define MAXDATASIZE     (256*1024)      /* bytes */

#endif ELT_DATA_H
```

```
/* elt_mon.h version 2.1, 7/25/88 */

/*
 * This file contains definitions for the Hypercube node monitor program
 */

#ifndef ELT_MON_H
#define ELT_MON_H

#include "elt_symbol.h"

struct cmdpacket {      /* command packet used to communicate with monitor */
   int message;
   char name[NAMESIZE];
   int ack;
};

/* values for cmdpacket.message */
#define READ_LOADMAP            0xFFF1/* read a node symbol's load map*/
#define READ_DATA               0xFFF2/* read a node symbol's data */
#define WRITE_LOADMAP           0xFFF3/* write a node symbol's load map*/
#define WRITE_DATA              0xFFF4/* write a node symbol's data */
#define CREATE_SYMBOL           0xFFF5/* create a symbol in the nodes */
#define DELETE_SYMBOL           0xFFF6/* delete a symbol from the nodes */
#define LOAD_MODULE             0xFFF7/* download a node module */
#define EXECUTE_MODULE          0xFFF8/* execute the current node module */
#define EXECUTE_BUILTIN         0xFFF9/* execute a builtin function */
#define REDIST_DATA             0xFFFA/* re-distribute data in the cube */
#define EXIT                    0xFFFB/* tell the node monitor to exit */

/* values for cmd.ack_wanted */
#define SENDACK             1
#define NOACK               0

typedef struct {        /* elt acknowledgement packet */
   int status;          /* status (OK, error, etc) */
   int message;         /* message being ack'ed */
} ackpacket;

/* values for ackpacket.status */
#define ELT_OK              0       /* OK return code */
#define ELT_ERROR           -1      /* error return code */
#define ELT_ACK(m,s)elt_ack.message=(m); \\
                    elt_ack.status=(s); \\    dumpelt(&elt_ack,sizeof(ackpacket));
extern ackpacket elt_ack;

#endif ELT_MON_H
```

```
/* elt_symbol.h version 2.1, 7/25/88 */

 * This file contains the definitions for the node symbol tables used
 * for hypercube data management
 */
#ifndef ELT_SYMBOL_H
#define ELT_SYMBOL_H

#define NAMESIZE            80      /* max length of symbol name */
#define MAXSYMS             100     /* max number of symbols */

typedef char elt_name[NAMESIZE];    /* type of symbol names */

struct elt_loadmap{
    int loadtype;                   /* load type if a simple type, -1 if unknown */
    int sb;                         /* starting band */
    int nb;                         /* number of bands */
    int sl;                         /* starting line */
    int nl;                         /* number of lines per band*/
    int ss;                         /* starting sample */
    int ns;                         /* number of samples per line */
};

struct elt_symbol{
    unsigned char *data;            /* pointer to data area */
    elt_name name;                  /* symbol name */
    int global;                     /* 1 if symbol is global, 0 if local */
    int datatype;                   /* data type code */
    struct elt_loadmap loadmap;     /* distribution pattern */
};

/* macros for use with pointers to symbol table entries */
#define DATA(s)             ((s)->data)
#define NAME(s)             ((s)->name)
#define GLOBAL(s)           ((s)->global)
#define DATATYPE(s)         ((s)->datatype)
#define LOADMAP(s)          ((s)->loadmap)
#define LOADTYPE(s)         ((s)->loadmap.loadtype)
#define SB(s)               ((s)->loadmap.sb)
#define NB(s)               ((s)->loadmap.nb)
#define SL(s)               ((s)->loadmap.sl)
#define NL(s)               ((s)->loadmap.nl)
#define SS(s)               ((s)->loadmap.ss)
#define NS(s)               ((s)->loadmap.ns)
#define ELEMENTSIZE(s)      (elt_elementsize[DATATYPE(s)])
#define NUMELEMENTS(s)      (NB(s)*NL(s)*NS(s))
#define DATASIZE(s)         (NUMELEMENTS(s)*ELEMENTSIZE(s))

/* global array containing sizes of various types */
extern int elt_elementsize[];       /* defined in elt_monitor.c */

/* values for elt_symbol.global */
#define LOCALSYM   0        /* symbol is known only in cube */
#define GLOBALSYM  1        /* symbol is known to both elt and cp */
```

```
/* definitions of data type codes
 * DO NOT CHANGE THESE WITHOUT MAKING SURE THE RESULTING CODES
 * ARE IDENTICAL ON BOTH THE CP AND ELT SIDES!
 * WHEN CHANGING THEM, BE SURE TO UPDATE THE INITIALIZATION FOR
 * elt_elementsize[] IN elt_init() TOO!
 * (CP data type codes defined in symbol.h)
 * (elt_init() defined in elt_monitor.c)
 */

#ifndef SYMBOL_H                    /* defined by symbol.h (on CP side only) */
/* included only on hypercube ELT side */
#define UNDEF_TYPE      0       /* undefined */
#define CHAR_TYPE       1       /* char */
#define SHORT_TYPE      2       /* short */
#define INT_TYPE        3       /* int */
#define FLOAT_TYPE      4       /* float */
/* #define STRING_TYPE  5       /* string - UNUSED IN CUBE */
/* #define BOOL_TYPE    6       /* Boolean - UNUSED IN CUBE */
/* #define KEYWORD_TYPE 7       /* keyword type - UNUSED IN CUBE */
#define DOUBLE_TYPE     8       /* double */
#define NUM_TYPES       8       /* number of types defined */
#endif SYMBOL_H

typedef struct elt_symbol SYMBOL;

#define NULLSYM ((SYMBOL *) 0)          /* null symbol pointer */
#define NULLD ((unsigned char *) 0)     /* null data pointer */

/* the symbol table */
extern SYMBOL symtab[];             /* defined in elt_symbol.c */

#endif ELT_SYMBOL_H
```

```
/* funclib.h version 2.1, 7/25/88 */

#ifndef FUNCLIB_H
#define FUNCLIB_H

#define FUNCNAMESIZE     32
#define MAX_FUNCTIONS    511
#define NO_FUNCTION      MAX_FUNCTIONS

typedef unsigned int functionnum;

struct cipe_function {
  char name[FUNCNAMESIZE];
  int args;
  int builtin;
  int result;
  int (*codeptr)();
  struct code_seg_def *segptr;
  char *text;
  int executing;
  char *pathname;
};

/* macros for use with pointers to symbol table entries */
#define CIPEFUNCTION(s)     ((s)->name)
#define CIPEARGS(s)   ((s)->args)
#define CIPEBUILTIN(s)      ((s)->builtin)
#define CIPERESULT(s)       ((s)->result)
#define CIPECODEPTR(s)      ((s)->codeptr)
#define CIPESEGPTR(s)       ((s)->segptr)
#define CIPETEXT(s)   ((s)->text)
#define CIPEEXECUTING(s)    ((s)->executing)
#define CIPEPATHNAME(s)     ((s)->pathname)

extern struct cipe_function *cipe_get_function_ptr();
extern struct cipe_function *cipe_function_index_to_ptr();
extern char *cipe_function_index_to_name();

#endif FUNCLIB_H
```

```
/* image_hdr.h version 2.1, 7/25/88 */

#ifndef IMAGE_HDR_H
#define IMAGE_HDR_H

/*
 * This file contains structure and constant definitions for
 * the CIPE header.
 */
#define CIPE_HEADER "CIPE"
#define NUMBER_LINES "number of lines"
#define NUMBER_SAMPLES "number of samples"
#define NUMBER_BANDS "number of bands"
#define CIPE_TYPE "type"
#define OFFSET "offset"
#define BYTE_STRING "byte"
#define INT_STRING "int"
#define FLOAT_STRING "float"
#define SHORT_STRING "short"

#define NUMBER_OF_BANDS_DEFAULT    1
#define NUMBER_OF_LINES_DEFAULT    -1/* no default */
#define NUMBER_OF_SAMPLES_DEFAULT -1/* no default */
#define OFFSET_DEFAULT             0
#define DATA_TYPE_DEFAULT          CHAR_TYPE

struct image_header_type {
       char    header_indicator[5];
       int     data_type,
               number_of_lines,
               number_of_samples,
               number_of_bands,
               offset;         /* offset in bytes where data starts. */
                               /* used to skip headers. */
        float    starting_wavelength,
              ending_wavelength;
       unsigned char   reserved[32];
        struct image_header_type *next;        /* for use when images */
                                               /* are in a linked list. */
};

struct image_header_type *get_header();

#endif IMAGE_HDR_H
```

```
/* menus.h version 2.1, 7/25/88 */

int cipe_setup(),cipe_help_function(),cipe_help_symbol();
int cipe_menu_read(),
    cipe_menu_write(),
    cipe_menu_assign(),
    cipe_menu_delete_symbol(),
    cipe_menu_builtin(),
    cipe_menu_application(),
    cipe_appl_convolve(),
    cipe_appl_frequency(),
    cipe_appl_contrast(),
    cipe_appl_size(),
    cipe_appl_tiept(),
    cipe_appl_project(),
    cipe_appl_template_match(),
    cipe_appl_feature_match(),
    cipe_appl_texture_class(),
    cipe_appl_spectra_class();

/* display */
int b_alloc(),c_alloc(),p_alloc(),ierase(),gerase(),lstretch(),tstretch(),
    histo(),zoom(),r_cursor(),w_cursor(),draw(),mssdraw(),mssplot(),plot3d();

struct func_name_pair func_table[]={
    {cipe_setup,"setup"},
    {cipe_help_function,"help_function"},
    {cipe_help_symbol,"help_symbol"},
    {cipe_menu_assign,"assign_data"},
    {cipe_menu_read,"read_image"},
    {cipe_menu_write,"save_image"},
    {cipe_menu_delete_symbol,"delete"},
    {cipe_menu_builtin,"builtin"},
    {cipe_menu_application,"user"},
    {cipe_appl_convolve,"convolve"},
    {cipe_appl_frequency,"frequency"},
    {cipe_appl_contrast,"contrast"},
    {cipe_appl_size,"size"},
    {cipe_appl_tiept,"tiept"},
    {cipe_appl_project,"project"},
    {cipe_appl_template_match,"template"},
    {cipe_appl_feature_match,"feature"},
    {cipe_appl_texture_class,"texture"},
    {cipe_appl_spectra_class,"spectra"},
    {b_alloc,"balloc"},
    {c_alloc,"calloc"},
    {p_alloc,"palloc"},
    {r_cursor,"rcursor"},
    {w_cursor,"wcursor"},
    {histo,"histo"},
    {lstretch,"lstretch"},
    {tstretch,"tstretch"},
    {ierase,"ierase"},
    {gerase,"gerase"},
```

```
    {draw,"draw"},
    {mssplot,"mssplot"},
    {plot3d,"plot3d"},
    {mssdraw,"mssdraw"},
    {zoom,"zoom"},
    {0,"/keep last"},
};
```

```
/* opcodes.h version 2.1, 7/25/88 */

#ifndef OPCODES_H
#define OPCODES_H

   /* instructions and program structure */

#define OPCODE_WID      6
#define FUNC_IND_WID    9
#define SYM_IND_WID     9
#define ATTR_WID        5
#define WSCMD_WID       3
#define STATUS_WID      3
#define EXECPART_WID    3
#define BOOL_WID        1
#define SCOPE_WID       1
#define MENU_MODE_WID   1

struct instruction {
   unsigned int opcode                  : OPCODE_WID;
   union {
        struct {
           symbolnum lhs                : SYM_IND_WID;
           symbolnum rhs                : SYM_IND_WID;
           symbolnum result             : SYM_IND_WID;
        } arith,rel,bool;
        struct {
           symbolnum src                : SYM_IND_WID;
           symbolnum dest               : SYM_IND_WID;
        } assign;
        struct {
           symbolnum arg                : SYM_IND_WID;
           symbolnum result             : SYM_IND_WID;
           struct indexlist_def *indices;
        } index;
        struct {
           functionnum func             : FUNC_IND_WID;
           struct exprlist_def *args;
           symbolnum result             : SYM_IND_WID;
        } call;
        struct {
           symbolnum loc                : SYM_IND_WID;
           symbolnum expr1              : SYM_IND_WID;
           symbolnum expr2              : SYM_IND_WID;
           symbolnum step               : SYM_IND_WID;
        } br,brz,brfor;
        struct {
           unsigned int attr            : ATTR_WID;
           symbolnum value              : SYM_IND_WID;
        } set;
        struct {
           unsigned int command         : WSCMD_WID;
           symbolnum ws                 : SYM_IND_WID;
        } wscmd;
```

```
        struct {
            char *cmd;
        } shellcmd;
        struct {
            char *name;
            struct code_seg_def *codeseg;
            struct arglist_def *args;
            unsigned int result;
            char *text;
            unsigned int scope          : SCOPE_WID;
        } define;
        struct {
            struct exprlist_def *exprs;
        } print;
        struct {
            functionnum func            : FUNC_IND_WID;
        } show;
        struct {
            symbolnum symbol            : SYM_IND_WID;
            symbolnum filename          : SYM_IND_WID;
        } read,write;
        struct {
            symbolnum image             : SYM_IND_WID;
            struct location_def location;
        } display;
        struct {
            unsigned int menumode       : MENU_MODE_WID;
        } menu;
    } fields;
};

extern struct code_seg_def *code_seg;
extern int num_code_segs;
extern struct instruction *curr_op;

#define NOP_CODE        0       /* NOP - no arguments*/
#define LT_CODE         1       /* RELationals - arg1, arg2, result*/
#define LTE_CODE        2
#define EQ_CODE         3
#define NE_CODE         4
#define GTE_CODE        5
#define GT_CODE         6
#define AND_CODE        7       /* BOOLeans - arg1, arg2, result*/
#define OR_CODE         8
#define BITAND_CODE     9
#define BITOR_CODE      10
#define BITXOR_CODE     11
#define PLUS_CODE       12      /* arithmetics - arg1, arg2, result*/
#define MINUS_CODE      13
#define MULT_CODE       14
#define DIV_CODE        15
#define GETS_CODE       16      /* GETS - source, destination*/
#define NOT_CODE        17      /* 1ARGument (unary) ops - arg, result*/
#define BITNOT_CODE     18
```

```
#define NEGATE_CODE      19
#define INDEX_CODE       20     /* SPECIAL - arg1, result, indices*/
#define CALL_CODE        21     /*     func, arglist*/
#define BR_CODE          22     /*     location*/
#define BRZ_CODE         23     /*     location, expr*/
#define BRFOR_CODE       24     /*     location, expr1, step, expr2*/
#define SET_CODE         25     /* SYSCMD - attr, value*/
#define WSCMD_CODE       26     /*     workspace name*/
#define BANG_CODE        27     /*     unix command*/
#define QUIT_CODE        28     /*     no arguments*/
#define STATUS_CODE      29     /*     display current status*/
#define DEFINE_CODE      30     /*     name, code seg ptr, arg list*/
#define PRINT_CODE       31     /*     expr*/
#define SHOW_CODE        32     /*     function index*/
#define FUNCTIONS_CODE   33     /*     no arguments*/
#define SYMBOLS_CODE     34     /*     no arguments*/
#define TRACES_CODE      35     /*     no arguments*/
#define READ_CODE        36     /*     expressions, filename*/
#define WRITE_CODE       37     /*     expressions, filename*/
#define DISPLAY_CODE     38     /*     symbol index*/
#define MENU_CODE        39     /*     no arguments*/
#define HELP_CODE        40     /*     no arguments*/

#define ISNOP(op)       (op == NOP_CODE)

#define ISREL(op)       (op >= LT_CODE && op <= GT_CODE)
#define ISBOOL(op)      (op >= AND_CODE && op <= BITXOR_CODE)
#define IS2ARG(op)      (op >= LT_CODE && op <= DIV_CODE)

#define ISGETS(op)      (op == GETS_CODE)
#define IS1ARG(op)      (op == NOT_CODE || op == BITNOT_CODE || op == NEGATE_CODE)

#define ISCALL(op)      (op == CALL_CODE)
#define ISINDEX(op)     (op == INDEX_CODE)
#define ISBRANCH(op)        (op == BR_CODE || op == BRZ_CODE || op == BRFOR_CODE)
#define ISSPECIAL(op)       (ISCALL(op) || ISINDEX(op) || ISBRANCH(op))

#define ISSET(op)       (op == SET_CODE)
#define ISWSCMD(op)         (op == WSCMD_CODE)
#define ISBANG(op)      (op == BANG_CODE)
#define ISQUIT(op)      (op == QUIT_CODE)
#define ISSYMBOLS(op)       (op == SYMBOLS_CODE)
#define ISFUNCTIONS(op)     (op == FUNCTIONS_CODE)
#define ISTRACES(op)        (op == TRACES_CODE)
#define ISSHOW(op)      (op == SHOW_CODE)
#define ISDEFINE(op)    (op == DEFINE_CODE)
#define ISPRINT(op)     (op == PRINT_CODE)
#define ISREAD(op)      (op == READ_CODE)
#define ISWRITE(op)     (op == WRITE_CODE)
#define ISDISPLAY(op)       (op == DISPLAY_CODE)
#define ISMENU(op)      (op == MENU_CODE)
#define ISHELP(op)      (op == HELP_CODE)
#define ISSYSCMD(op)        (op >= SET_CODE && op <= HELP_CODE)
```

```
struct builtin_def {
    char *name;
    int (*type_check)();
    int (*exec_code)();
};

extern int cipe_lt_builtin(),cipe_lte_builtin(),cipe_eq_builtin(),
    cipe_ne_builtin(),cipe_gte_builtin(),cipe_gt_builtin();
extern int cipe_and_builtin(),cipe_or_builtin(),cipe_bitand_builtin(),
    cipe_bitor_builtin(),cipe_bitxor_builtin();
extern int cipe_plus_builtin(),cipe_minus_builtin(),cipe_mult_builtin(),
    cipe_div_builtin();

extern int cipe_int_float(),cipe_int_only();

extern struct builtin_def builtins[];

    /* workspace commands */

#define LOADWS      0
#define SAVEWS      1
#define EDITWS      2

extern char *wscmds[];

#endif OPCODES_H
```

```
/* symbol.h version 2.1, 7/25/88 */

#ifndef SYMBOL_H
#define SYMBOL_H

#define SYMBOL_NOT_FOUND 0x00
#define SYMBOL_IN_CIPE    0x01
#define SYMBOL_IN_CUBE  0x02
#define SYMBOL_IN_FILE  0x04
#define CIPENAMESIZE 80

#define MAX_SYMBOLS      511
#define NO_SYMBOL MAX_SYMBOLS

struct loadmap {
    int loadtype;
    int sb;
    int nb;
    int sl;
    int nl;
    int ss;
    int ns;
};

struct cipe_symbol {
  unsigned char *data;
  char name[CIPENAMESIZE];
  char file[CIPENAMESIZE];
  unsigned int ndim;
  unsigned int sb;
  unsigned int nb;
  unsigned int sl;
  unsigned int nl;
  unsigned int ss;
  unsigned int ns;
  unsigned int datatype;
  unsigned int loadtype;
  struct loadmap *cubeload;
};

extern int cipe_size[];

/* macros for use with pointers to symbol table entries */
#define CIPEDATA(s)  ((s)->data)
#define CIPENAME(s) ((s)->name)
#define CIPEFILE(s)    ((s)->file)
#define CIPENDIM(s)  ((s)->ndim)
#define CIPESB(s)      ((s)->sb)
#define CIPENB(s)     ((s)->nb)
#define CIPESL(s)      ((s)->sl)
#define CIPENL(s)      ((s)->nl)
#define CIPESS(s)     ((s)->ss)
#define CIPENS(s)     ((s)->ns)
#define CIPEDATATYPE(s)    ((s)->datatype)
```

```
#define CIPELOADTYPE(s)     ((s)->loadtype)

#define CIPELOADMAP(s) ((s)->cubeload)
#define CIPEELEMENTSIZE(s) cipe_size[CIPEDATATYPE(s)]
#define CIPENUMELEMENTS(s) CIPENL(s)*CIPENS(s)*CIPENB(s)
#define CIPEDATASIZE(s) CIPENUMELEMENTS(s)*CIPEELEMENTSIZE(s)

/* function types */
extern struct cipe_symbol *cipe_get_symbol_ptr();
extern struct cipe_symbol *cipe_symbol_index_to_ptr();
extern char *cipe_symbol_index_to_name();

typedef unsigned int symbolnum;

#define UNDEF_TYPE 0
#define CHAR_TYPE  1
#define SHORT_TYPE 2
#define INT_TYPE   3
#define FLOAT_TYPE 4
#define DOUBLE_TYPE     5
#define BOOL_TYPE  6
#define STRING_TYPE     7
#define KEYWORD_TYPE    8

#define NUM_TYPES  9

#endif SYMBOL_H
```

TECHNICAL REPORT STANDARD TITLE PAGE

1. Report No. 88-32	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle CONCURRENT IMAGE PROCESSING EXECUTIVE (CIPE)		5. Report Date 10-1-88 6. Performing Organization Code
7. Author(s) Meemong Lee		8. Performing Organization Report No.
9. Performing Organization Name and Address JET PROPULSION LABORATORY California Institute of Technology 4800 Oak Grove Drive Pasadena, California 91109		10. Work Unit No. 11. Contract or Grant No. NAS7-918 13. Type of Report and Period Covered JPL Publication
12. Sponsoring Agency Name and Address NATIONAL AERONAUTICS AND SPACE ADMINISTRATION Washington, D.C. 20546		14. Sponsoring Agency Code RE150 BP-889-20-41-17-62

15. Supplementary Notes

16. Abstract

This report describes the design and implementation of a Concurrent Image Processing Executive (CIPE), which is intended to become the support system software for a prototype high performance science analysis workstation. The target machine for this software is a JPL/Caltech Mark IIIfp Hypercube hosted by either a MASSCOMP 5600 or a Sun-3, Sun-4 workstation; however, the design will accommodate other concurrent machines of similar architecture, i.e., local memory, multiple-instruction-multiple-data (MIMD) machines. The CIPE system provides both a multimode user interface and an applications programmer interface, and has been designed around four loosely coupled modules; (1) user interface, (2) host-resident executive, (3) hypercube-resident executive, and (4) application functions. The loose coupling between modules allows modification of a particular module without significantly affecting the other modules in the system. In order to enhance hypercube memory utilization and to allow expansion of image processing capabilities, a specialized program management method, incremental loading, was devised. To minimize data transfer between host and hypercube a data management method which distributes, redistributes, and tracks data set information was implemented. The data management also allows data sharing among application programs. The CIPE software architecture provides a flexible environment for scientific analysis of complex remote sensing image data, such as imaging spectrometry, utilizing state-of-the-art concurrent computation capabilities.

17. Key Words (Selected by Author(s))	18. Distribution Statement
Geosciences and Oceanography (General), Computer Programming and Software, Space Sciences (General)	Unclassified--Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
UNCLASSIFIED	UNCLASSIFIED	200	